AMIGA RAY TRACING IN C—PART I

AC's TECH AMIGA

For The Commodore

Volume 1 Number 4
US $14.95 Canada $19.95

AREXX APPLICATIONS

- GPIO—Low-Cost Sequence Control
- Programming with the ARexxDB Records Manager
- Programming the Amiga's GUI in C—Part III
- Using Interrupts for Animating Pointers
- STOX—An ARexx Based System for Maintaining Stock Prices
- Language Extensions—Strings of Type StringS
- State of Amiga Development Denver DevCon Address

Plus

Build your own variable rapid-fire joystick

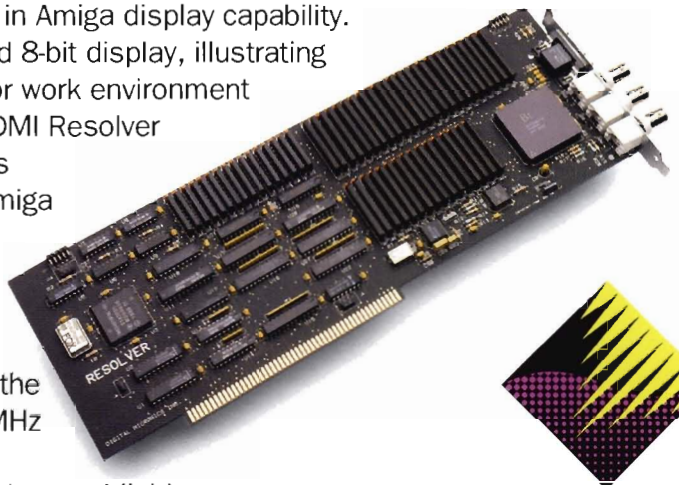
0 74470 78640 5 04

Don't be Fooled by any other Solution. 1280x1024 Resolution.

DMI Resolver™

- 1280x1024 Resolution
- 8-bit Color Graphics
- 16-million Color Palette
- 60MHz Processor
- Programmable Resolution

The DMI Resolver™ graphics co-processor board offers a new dimension in Amiga display capability. Shown above is an unretouched 8-bit display, illustrating the 1280x1024 resolution color work environment provided by the Resolver. The DMI Resolver boosts the display and graphics processing capabilities of all Amiga A2000 and A3000 series computers, under both AmigaDOS and UNIX operating systems. Not to be confused with a frame buffer or grabber, the Resolver is a lightning fast 60MHz graphics co-processor. Whatever your application – desktop publishing, presentation graphics, animation, 3D modeling, ray tracing, rendering, CAD – let the Resolver move you into a new realm of resolution and workstation quality display.



Digital Micronics, Inc.
5674 El Camino Real, Suite P
Carlsbad, CA 92008
Tel: (619) 431-8301 • FAX: (619) 931-8516
Call for more information and the dealer nearest you.

State of Amiga Development Denver DevCon Address

Jeff Scherb is an important individual in the Amiga community. As vice president of CATS, Jeff has a range of responsibilities for creating and supporting new products and developers in the Amiga market. The following is his keynote address from Denver DevCon '91 (September 3 through 7).

A lot has happened since we were together last year at the Atlanta Devcon. I spent a great deal of time thinking about this year's keynote address, and in going over the events of the past year, I was very surprised to see how much we at Commodore and you as developers have accomplished. We often lose sight of the progress of the war because of our particular daily battle, and I think it's worthwhile to take a few minutes to review what we've accomplished over the last year.

It's Been A Good Year

We are now getting the press coverage we deserve. The Amiga is finally being recognized as a powerful and cost-effective alternative to the PC and Macintosh. We are getting noticed by the professional computing community.

The A3000 is shipping in volume. The recent "A3000 Power Up" sales promotion run by Commodore in the U.S. has exceeded all of our expectations, and has, in fact, resulted in a backlog of orders for the machine.

The A500 continues to gain momentum, particularly in Europe. Many believe that the 500 is now poised to repeat the success of the Commodore 64.

CDTV is shipping to tremendous press reviews. Over 100 titles are already available for the machine. The press now defines the "interactive multimedia player" machine generically as CDTV, rather than CD-I as it did a year ago. We are now the leader that the "other guys" have to follow.

We are now shipping Amiga running AT&T System V, Release 4 UNIX. This is one of the earliest commercial implementations of V.4, and our implementation includes Open Look and color X Windows running on the University of Lowell graphics card. Over 150 applications are already available for this machine.

Other Accomplishments

The A3000T "tower" machine is now shipping. This elevates Amiga expandability and performance to new levels.

AmigaDOS 2.0 is now finished and ROMS are being manufactured.

Over 1000 developers are registered in the CATS-US Developer support program. Over 500 are registered in the support program in Europe. This is a significant increase over last year.

Commodore continues to be profitable, and just closed the fiscal year at over $1 billion, for the first time since the '64 days.

In the next month or two, we expect to ship the three-millionth Amiga!

So we all have good reason to reflect back on the year since the last developer's conference and feel very good about our accomplishments.

Jeff Scherb giving his Keynote Address at the Denver DevCon '91.

Present

Let's turn to the present. Where should developers focus their efforts to make 1992 an even better year?

First, make sure you are strong in the areas of the world where Commodore is strong. You will see from our annual report that Commodore does about 85 percent of its business in Europe. If you don't have good European distribution, or your product is not available in the local European languages, you are missing out on a very large market.

Support AmigaDOS 2.0 now! The time has come. We will be shipping machines with 2.0 in ROM very soon, and the time for you to release 2.0 compatible upgrades is now, if you haven't done it already.

Continue your focus on quality. In general, Amiga software is of much better quality now than it was a year ago, but we all need to continually focus on quality.

(continued on page 96)

AC's TECH
MessagePort

Dear *AC Tech*:

I'm always pleased to find that Paul Castonguay will stand up for and publicize the advantages of all the programming languages. Well, let's only *think* about COBOL, since there's no compiler available ayway, thank goodness. At the same time, Paul has done an outstanding job over the years in bringing much esoteric stuff to the unwashed public. It's the good old rule: good programmers make good programmers. I have had to depend on columnists like Paul. It's a long way to any help out here in the country, so that I always look for his byline in the index of *AC Tech.*

His series on C is starting at the proper level and proceeding at the proper pace. I look forward to a few more words on the nuts and bolts of the SAS C compiler, such as, what's a pragma, and what's it an acronym for? How does one select "includes" from a mountain of files, and so on?

His letter on sorting prompted me to get busy wih my pet, F-BASIC. For the bubblesort, it weighed in at two min., 35 secs. The combsort required six seconds. I finally got down to coding my old quicksort routine in F-BASIC and got four seconds on that one.

Keep up the good work on *AC's Tech.*

Homer C. Waits
Columbia, VA 23038

I'm very much impressed at the speed of your pet, F-BASIC, on the combsort algorithm. True BASIC often has a hard time competing in such situations because it doesn't allow the programmer to control the way the computer does arithmetic. It offers only one numeric data type, converting numbers from integer to floating point as it sees fit. And there's no way to choose between byte size, word size, or long-word-size integer arithmetic. But to tell the truth, I find that a blessing. Having to deal with how numbers are actually stored in a computer is another one of those nasty technical details that divert a programmer's attention from solving a particular algorithm. I guess I'm a high-level programmer at heart. Of course, once a solution has been achieved, a great deal of pleasure can be derived by tweaking it for maximum speed. I'm happy to hear that my article has inspired some of that inquisitive work.

Your question about #pragma is a good one. There are published definitions, like "An implementation-defined preprocessor instruction," but that wasn't your question. You would like to know the origin of the word. Oddly enough, I haven't been able to find anyone who knows. SAS/C didn't know even though it's used in their product. The employee who might know was in Europe at the time. It's natural enough to guess that it has to do with the word pragmatic, but that's only a guess. It could just as easily mean "Please Read and Assemble Gently My Algorithm," or Program to Regurgitate And Garble My Abacus."

You question is a welcome breath of reality in this fast-changing world of techno-babble. Perhaps there are some readers who can help.
—Paul Castonguay

Dear *AC Tech:*

I have enjoyed playing with Paul Castonguay's CombSort programs. I have to report that there is a logical error in Comb_Sort.bas. In the subroutine Sort.Values the line

```
WHILE(switch <> 0 AND gap <> 1)
```

should read

```
WHILE(switch <> 0 OR gap <> 1)
```

The AND form quits before the array is fully sorted.

I have two suggestions for articles that I would like to see you publish in the future. First, I would like to see an explanation of how program loading works and how hunks are organized. The second concerns the console.device and the utility SetMap. The 1.3 version of SetMap does two things when run from the CLI: it changes the keymap of the CLI window that it is run from, and it changes the default keymap that the console.device will use in any console window that is opened subsequently. However, it does not change the keymap in any other console window that already exists. I would like to have an improved SetMap that changes the keymap in all console windows. The change is simple enougjh to make; you need only send a CD SETKEYMAP command to every ConUnit. To do this, you need to know the address of every ConUnit—and this is where I'm stymied. Presumably, the console.device maintains a list of its ConUnits, but I can't find any documentation on this. If one of your experts could explain how the console.device keeps track of its ConUnits and could perhaps write an improved SetMap as an illustration, I would be grateful.

Douglas Nelson
Omaha, NE 68104

I confess, it's my fault. I made the logic error when translating the DO-UNTIL loop construct of True BASIC to the WHILE-WEND of AmigaBASIC and I failed to notice that upon completion of the program certain few dots were still out of order. Boy, I guess I'll have to clean my eyeglasses more often. Congratulations on finding the error.

I share your desire to see good instructional material on both the console.device and the organization of hunks. I myself do not program the Amiga at such low levels, but I certainly encourage anyone who does to share his or her knowledge. How about somebody submitting an article?
—Paul Castonguay

Send your questions and comments to:

AC's TECH MessagePort
P.O. Box 869
Fall River, MA 02722-0869

Readers whose letters are published will receive five public domain disks free of charge. (All letter become the property of P.i.M. Publications, Inc. The AC's TECH editors reserve the right to edit all letters for length and clarity.)

Contents

Volume 1, Number 4

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
One Currant Place
Fall River, MA 02722

AC's TECH/AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Corporate Trainer:	Virginia Terry Hicks
Traffic Manager:	Robert Gamble
International Coordinator:	Donna Viveiros
Marketing Manager:	Ernest P. Viveiros Sr.
Programming Artist:	E. Paul

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Ernest P. Viveiros, Jr.
Associate Editor:	Jeffrey Gamble
Hardware Editor:	Ernest P. Viveiros Sr.
Technical Editor:	J. Michael Morrison
Technical Associate:	Aimée B. Abren
Copy Editors:	Timothy Duarte Paul Larrivée
Video Consultant:	Frank McMahon
Art Director:	William Fries
Photographer:	Paul Michael
Illustrator:	Brian Fox
Research & Editorial Support:	Melissa A. Torres
Production Assistant:	Valerie Gamble

ADVERTISING SALES

Advertising Manager:	Donna Marie
Advertising Associate:	Wayne Arruda

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

SPECIAL THANKS TO:
Richard Ward & RESCO

AC's TECH For The Commodore Amiga™ (**ISSN 1053-7929**) is published quarterly by PiM Publications, Inc., One Currant Road, P.O. Box 869, Fall River, MA 02722-0869.

Subscriptions in the U.S., 4 issues for **$44.95**; in Canada & Mexico surface, **$52.95**; foreign surface for **$56.95**.

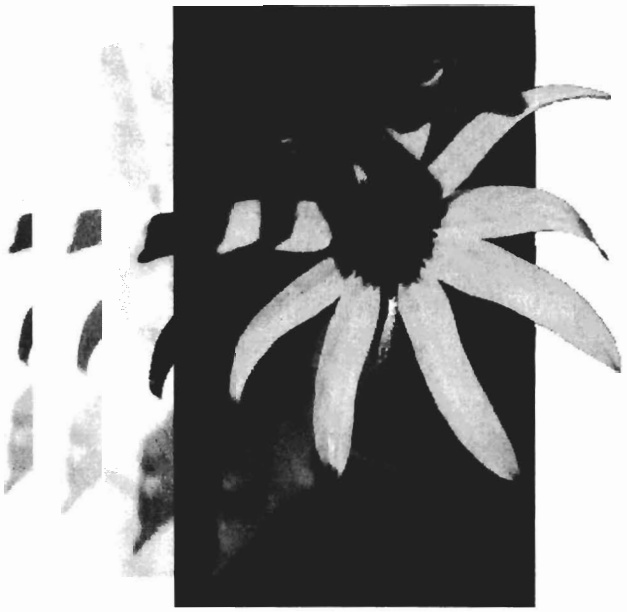
Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 869, Fall River, MA 02722-0869. Printed in the U.S.A.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.



High Resolution Output

from your AMIGA™ DTP & Graphic Documents

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2450 dpi (up to 154 lpi) at a extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

Who are we? We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.

For specific format information, please call.

For more information call 1-800-345-3360

Just ask for the service bureau representative.

GPIO

The author and publisher assume no liability for damage resulting from the use of the software and/or hardware described in this article and makes no warranty either expressed or implied as to its performance. All software and hardware, however, have been used by the author as described.

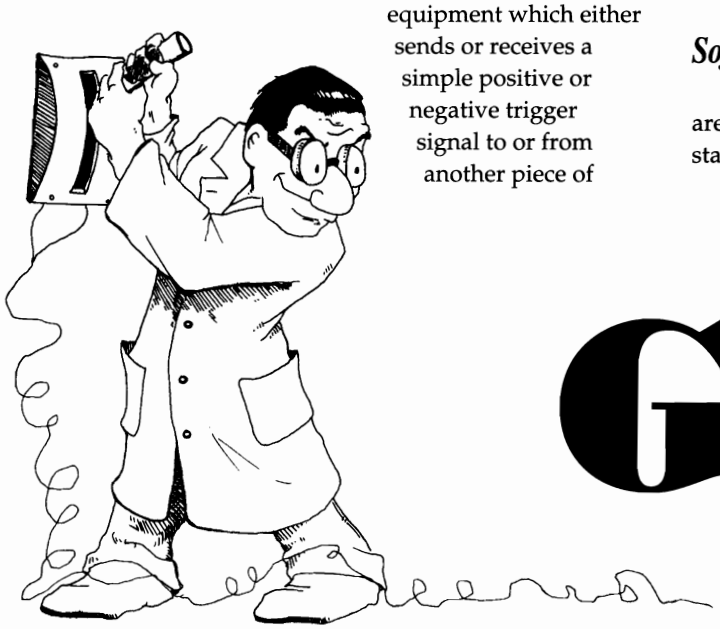
The software described here is not public domain but is freely distributable. The author, Ken Hall, retains all rights to publication and modifications. Please leave the copyright notice and author's identification intact in all modules.

Now, with that out of the way....

So you have tons of great software, some decent video hardware and a great idea for a presentation. All you need is a way to control these things at a precise time either automatically or manually. You are not alone and this easy project will go a long way toward providing that control, even sequencing for NewTek's Video Toaster. In fact, the Toaster is what inspired this project.

It is great that the Toaster includes an ARexx port. It is also great that it includes GPI capability. Unfortunately there is no way to activate the GPI from an ARexx script or to pause the script until a GPI pulse is received.

This article presents two relatively simple programs and optional hardware to add GPI (General Purpose Interface) capability to your Amiga. In case the GPI acronym is unfamiliar to you, it is a feature provided on many types of video equipment which either sends or receives a simple positive or negative trigger signal to or from another piece of equipment. Many video edit controllers have GPI outputs and can be used to start a player, audio deck, etc. , at a precise time.



The software presented here is called GPIO since it is capable of both General Purpose Input and Output. The optional hardware will provide for one input and two outputs. The hardware is optional because if all you want is a manual input trigger, then a common Amiga compatible joystick is all you need, à la a slide projector. The hardware is very simple and its sole purpose is to separate your Amiga and the equipment it is connected to. Since the Amiga game port (the one the mouse does not use) connects directly to a custom chip, this buffering is extremely desirable in the event of a mishap in a cable or the equipment. Additionally, the buffer adds more drive than one might expect the custom chip, Paula, to provide.

As mentioned previously, there are two pieces of software. Both provide the same essential capabilities (GPIO) but one has an ARexx port and the other does not. If you do not own ARexx by William S. Hawes, then the ARex version will not do you any good. Once version 2.0 of the Amiga O.S. is released, every one who buys it will get ARexx (unless the powers at Commodore decide differently.) So let's get down to business. Both versions of the software will be presented first, then the hardware. After that, a tutorial on how to use it will be provided. I should mention that I am self-taught in C and the Amiga O.S. In view of this there may be ways unknown to me that this software could be optimized. If such discoveries are made, I would very much appreciate your sharing them.

Software

Both versions are designed to be used from scripts, which are text files containing the sequence of commands to issue for starting various programs and talking to GPIO. If you prefer,

the ARexx version uses ARexx scripts or programs. The non-ARexx version is used from scripts run by the Amiga's 'execute' command. Both versions work by preventing the script from continuing until GPIO is done with its operation.

The commands provided by the non-ARexx version are as follows:

COMMAND	DESCRIPTION
GPIO p	Tell GPIO to wait for a positive level.
GPIO n	Tell GPIO to wait for a negative level.
GPIO 1	Send a pulse out on output #1.
GPIO 2	Send a pulse out on output #2.
GPIO 3	Send a pulse out on both outputs.
GPIO	Displays the above command syntax and my copyright notice.

The commands provided by the ARexx version are as follows:

COMMAND	DESCRIPTION
GPI p	Wait for a positive level.
GPI n	Wait for a negative level.
GPO 1	Send a pulse out on output #1
GPO 2	Send a pulse out on output #2
GPO 3	Send a pulse out on both outputs.
POLARITY up	Set both outputs high. Use this if your equipment wants negative going trigger pulses. This is the default.
POLARITY down	Does the opposite of 'POLARITY up'.
TIMEBASE n	Use this to tell the GPI routine how often to check the input for the commanded level where 'n' is a decimal number from 1 to 9. The interval is computed as n * 500 micro seconds.
VERSION	Display this syntax and my copyright notice.
EXIT	Tell GPIO_Rx to terminate.

You may be wondering about the polarity and timebase commands and why the non-ARexx version does not have them. The ARexx version stays running until it is told to terminate. This means that all variables can stay intact between commands. Once the polarity is set, it will remain stable between commands. The same is true for the timebase command but quite honestly, it could have been left out. It was left in just in case a good use is found. If you tried to use a polarity command in the non-ARexx version, it would work but as soon as the resources were returned to the Amiga, the port's pins would always go high. To get around this, we must include switches in the hardware to select polarity. These switches also provide a way to get around a limitation of the polarity command. POLARITY affects both output pins, so if you have one device that wants a negative input and another that wants a positive input, then the POLARITY command would be undesirable.

Now that you know what commands the programs provide, let's see how the programs work. The non-ARexx version (GPIO) will be presented first since it is the simplest. Following that, the ARexx version (GPIO_Rx) will be explained. Both are quite similar, but there are enough differences to warrant two explanations.

Theory of Operation

GPIO:

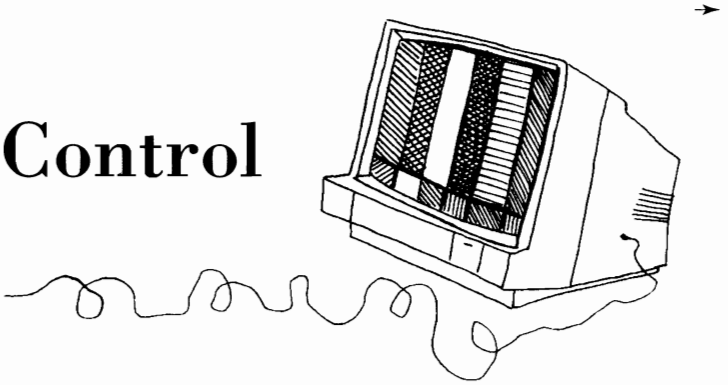
Please refer to Listing 1 (gpio.h), Listing 2 (gpio_init.c) and Listing 3 (gpio.c) during this explanation.

gpio.h contains the include files and definitions common to both versions. Not much else to tell here.

gpio_init.c contains the initialization and termination routines needed by both versions.

Low Cost Sequence Control

by Ken Hall



Startup and Initialization

main( ) *Listing 3*

Since GPIO is executed every time a command is used, the command line must be parsed for the command. Parsing is an ambitious term in this case since all commands are single character and only one is recognized per program execution. After parsing, the command is converted to upper case and the initialization routine is called. (The conversion to upper case eliminates case sensitivity problems)

gpio_init( ) *Listing 2*

This routine acquires the Timer used by the input and output trigger routines and initializes it. If the Timer can't be obtained the program exits with the appropriate message. Once this is done, the output port is allocated for GPIO use via the get_bits( ) routine and the port bits are initialized to high by writing all ones to the port. As long as the get_bits( ) routine returns, gpio_init will return the Timer signal bit to main( ).

get_bits( ) *Listing 2*

As documented in the Amiga Hardware Manual, you can use pins 5 and 9 of the game port for digital input and/or output. In order to do this reliably, one must acquire the pins from the O.S. by first opening the PotGo resource then allocating the bits. As in any Amiga function call, you had better not assume success on any allocation request. This is done by comparing what AllocPotBits( ) returned against what we asked for. If there is a difference, then the program will terminate with an error message. A thorough explanation of the bits will be provided later. For now, note POTBITS and its definition of 0xF000L. The bits (12 -15) are represented by the 'F' and are the bits being allocated to us. The lower case 'potbits' holds the actual bits allocated.

The Primary Functions

Now that we have the Timer and port bits for our use, we are back in the main( ) routine. A switch statement is entered where the previously "parsed" command is acted on. A command of 1, 2 or 3 will pass control to the output routine out( ). A command of P or N invokes the input routine in( ). If no command or an invalid command is found, the syntax message will be displayed along with the copyright notice and GPIO will quit.

out( ) *Listing 3*

The command is passed to this routine where another switch statement determines the action to take. Each case clause simply sets up two variables, y and z, to the values needed to drive the port pin(s) down then up. The variable known as 'timebase' is set to 500 for use by the send_timer( ) routine. This controls the output pulse width(s). The active level is written (low), the timer is started and the Amiga Wait( ) function is called with the Timer signal bit as its argument. When the Wait( ) function gets the Timer signal, it will return. The message is retrieved, the port pins are set inactive (twice for good measure) and the out( ) routine returns to main( ) where GPIO will terminate.

in( ) *Listing 3*

As usual, another switch statement is entered where the command is sorted out. The appropriate case clause sets a character string variable to the appropriate message, and a flag (GPI_flag) is set to indicate the input polarity to look for. If by some weird occurrence this routine is called without an argument, GPIO will terminate with no message. Assuming we did get a valid argument, the timer is started, the previously set message is printed and in( ) exits back to main( ).

We will assume that the in( ) routine was called so the ImAlive flag is set which will keep a while loop going until the input trigger is received. The timer is now going so we will Wait( ) for a signal from it or a control-F signal should the user decide not to wait. Once one of the signals is received, it is tested for which one. The control-F will cause GPIO to terminate with a "BREAK EXIT" message. Otherwise the message is retrieved, the input port pin 6 (bit 7) is separated by masking then tested by exclusive ORing with the polarity flag set in the in( ) routine. If the exclusive OR resulted in a '1', the input trigger was received and we break out of the while loop and terminate GPIO . If the exclusive OR computed to '0', the timer is restarted and the while loop continues. A word about the address use for checking pin 6 of the game port. The address 0xbfe001 is the port on CIA chip U300 where pin 6 attaches. I have not been able to find another way to do this since one cannot both allocate the pot bits and use the Amiga's GamePort.Device at the same time. This is the real reason for using the Timer—to tell us to check the pin and avoid using a busy-wait loop. I wanted to have triggering interrupt-driven rather than polled but so far this is the closest way I have found to do it. Anyone have any input on this?

Lets back-track a little now and talk about the send_timer( ) routine referred to in the out( ) and in( ) routines.

send_timer( ) *Listing 2*

This is a very simple routine which sets up the Timer request structure initialized in gpio_init( ) and calls the SendIO( ) function. The 'seconds' field is set to 0 and the 'microseconds' field is set to our variable 'timebase'. In a single tasking computer this would be the exact amount of time before the Timer returned its signal. The Amiga, however, is multitasking so the time interval is unfortunately at the mercy of how busy the system is as well as whether the

machine is accelerated or uses a stock 7 megahertz 68000. One thing is certain; the timer will not return in less time than requested. To simplify things, we just ask for a suitably small amount of time, 500 micros for output and 500 micros for input.

Termination

Most of the termination takes place in routines contained in the gpio_init.c module and are used by both versions. The cleanexit( ) routine is the only difference. After all, we will have to shut down the ARexx port in the GPIO_Rx version.

cleanexit( ) *Listing 3*

The termination message, if any, is printed, the Timer is returned to the system and the port bits are returned to the system. The generic exit( ) routine is then called.

kill_time( ) *Listing 2*

In the event a control-F was received while waiting for an input trigger, AbortIO( ) will gracefully terminate any Timer activity. The Timer.Device is then closed and its port is deleted.

kill_bits( ) *Listing 2*

This frees the port bits allocated in the get_bits( ) routine.

That does it for version 1, GPIO. Now on to GPIO_Rx.

GPIO_Rx:

Listing 1 (gpio.h) and Listing 2 (gpio_init.c) are the same for both versions. Three new modules will be explained here. The similarities and differences between/from GPIO will be pointed out. Please refer to Listing 4 (gpio_rx.c), Listing 5 (gpio_rexx.c) , and Listing 6 (gpio_rexx.h) as appropriate.

GPIO_Rexx.h *Listing 6*

This module contains all definitions, include files and the special global structure used to define ARexx commands.

Startup and Initialization

main( ) *Listing 4*

The same Timer and bits initialization is performed with the Timer signal bit being returned if all went well. Next an ARexx port must be allocated if anything is to work.

callRexxPort( ) *Listing 5*

As a quick overview, we look for a port with the name we wish to use (GPIO.rxport in this case sent from main( )). If the port is not found, it is created. The signal bit for ARexx message detection is initialized, commands to recognize are defined, and an address pointer to the routine to call to execute an ARexx sent command is initialized. The commands that we will recognize are defined in an array of structures whose address was sent to this routine from main( ) (the ARexx version of main( ) from now on.) There are two fields in these "RxCmdList" structures. The first is the upper-case character string to match to the command received from ARexx. The second field contains the address of the appropriate function for handling that command. It is important to realize that the array containing these structures must be terminated with a NULL structure pointer or the command search function will not know when the end of the valid command list has been reached. The array must also begin at element 0 with a valid structure, and each consecutive element must have valid fields in its structure. More on these structures when we follow the commands through.

We are back in the main( ) routine (listing 4) now with an operative ARexx port whose signal bit was returned from the callRexxPort( ) routine. If the signal is 0, there was an error during allocation so GPIO_Rx will quit. The version( ) routine will be called just to let us know all is well and remind us of the correct syntax. The 'ImAlive' variable was pre-initialized to TRUE so the while loop will be entered and the Wait( ) function called with the ARexx signal bit and control-F as the signals to wait on. Wait( ) will do just that until either signal is detected, at which time we find out which signal was received.

Just like GPIO, the control-F signal indicates user termination and that is what will occur. An ARexx signal will indicate that a message is waiting in our ARexx port for processing so the answerRexxPort( ) routine is called.

answerRexxPort( ) *Listing 5*

No arguments are sent to this routine since the ARexx port will have all we need and everything else was pre-initialized as global type variables. We verify that the rexxPort is not equal to 0. The variable 'ArgParsePointer' is set to the RexxMsg field rm_Args[0] where the command should be that our script sent. Both results fields for the ARexx reply are set to 0 for now. This indicates success in command execution once the ARexx message is replied to. It is the deferment of the reply which will prevent the script from executing further commands and what makes GPIO_Rx work. The 'pendingRexxMsg' variable is set to the address of the current message just in case a control-F termination request is received later while waiting on an input trigger. Now we must see if the command sent is in our command list. This is done with a combination of a 'for' statement and the check_command( ) routine. The 'for' statement begins at element 0 of the RxCmdList array of structures and passes that structure's name field address along with the 'ArgParsePointer' variable' to check_command( ) and will continue until check_command( ) returns a zero value or the end of the command list is reached.

→

check_command( ) *Listing 5*

The two character-string address pointers sent are used to compare the received command's characters, one by one, to the current RxCmdList's name field's characters, one by one. Each character of the received command is converted to upper case prior to comparison so we don't have to worry about case sensitivity. Notice that the commands in the command list are all in upper case. If any character does not match before the end of the valid command string is reached, that character is returned. Otherwise the NULL string terminator is returned to signal a command match.

If check_command( ) did not find a match after traversing the entire structure array, an error message indicating an unknown command is printed. At this point we will reply to ARexx. There is a compiler switch called 'COMPLAIN' which you may #define. If defined, the rm_Result1 field of the REXX reply message will be set to RXERRORNOCMD. Do this if you want the ARexx script to terminate on a GPIO ARexx error. If not defined, then ARexx will continue to execute the script being none the wiser about an error. The choice is yours. No harm will be done.

If a command match was found, then the routine pointed to by the address pointer field of our matching commands structure will be passed to the routine whose address is contained in the variable 'host'. The 'host' variable is used instead of hard coding the function so you can easily use this ARexx message handler in programs of your own design. The routine pointed to by 'host' is rexx_host( ) and lives in the gpio_rx.c module.

rexx_host( ) *Listing 4*

This routine processes a character pointer to obtain the commands arguments such as the output port number, input polarity, etc. This pointer is the same ArgParsePointer used in answerRexxPort( ); we just start the examination at the point in the string equal to the number of characters in the command. This is what the syntax; ArgParsePointer + strlen(RxCmdList->function_name) in answerRexxPort( ) does. The argument 'msg' is the RexxMsg received and is passed to any command execution function along with the parsed out command argument. It is important to realize that when you use function pointers, the number and type of arguments must be the same for each function referenced by a pointer. If not, you can get the compiler confused and the chance for goofing up a function is high.

Even if only one function uses a particular argument, write every function that might be called via a function pointer to expect the same arguments and all in the same order. To illustrate, let me explain that the version routine needs no arguments but the input routine (in( )) needs them. We could write the version function so it does not expect any but since it is called by rexx_host( ) exactly like in( ) is called, it will receive them anyway. Let's just be consistent. Anyway, back to executing commands.

We were at the point of calling the function (or routine) held in the RxCmdList structure where a command match was found. The processing of each command differs so each will be covered individually. The easy ones are first.

gpio_exit( )

Invoked by the EXIT command *Listing 4*

Sets the 'ImAlive' variable to zero and returns. Once the ARexx message is replied to and the while loop in main( ) is re-entered, the while loop will exit because of this.

version( )

Invoked by the VERSION command *Listing 4*

Just like its GPIO counterpart, GPIO_Rx syntax is printed along with the copyright notice. I don't know why you might want to call this from a script, but it is here if you do want to.

set_time( )

Invoked by the TIMEBASE n command. *Listing 4*

The 'p' argument will contain a single digit 1-9 in ASCII format. The upper half of the byte is masked off which converts it to decimal. If the result is 0, then 1 is assumed. Zero micro seconds will choke the Timer and the Guru will visit you. You may notice that we don't test for an ASCII numeric character. If you want to send it something else, it won't care but it is up to you to know what number will be left when the upper four bits are masked out. Whatever is left will be multiplied by 500 and stored in the 'timebase' variable.

polarity( )

Invoked by the POLARITY command *Listing 4*

This guy is intertwined with the out( ) function. For now just understand that the 'polar' variable is set to reflect the commanded polarity. The output pins are then driven to the inactive levels for that polarity, high for the 'U' polarity and low for 'D'.

out( )

Invoked by the GPO command *Listing 4*

The objective here is to create a pulse ONLY on the pin commanded, pin 5 for output 1 and pin 9 for output 2. The unaffected pin must stay in its inactive state, high for polarity 'U' and low for polarity 'D'. This is the best place to get serious about understanding the port pins and the relationship to the bits allocated.

The bits were mentioned briefly in the initialization phase where they were allocated in get_bits( ). A single register in the Paula chip is used to control pins 5 and 9 of the game port (others also, see the RKM libraries and devices and the Amiga Hardware Manual).

Two bits in this register are used to control each pin. Bits 12 & 13 control pin 5 and bits 14 & 15 control pin 9. Bits 13 and 15 control the direction of pins 5 & 9 respectively, 0 = input, 1 = output. Bits 12 & 14 control the state of the pins. Since GPIO uses the pins for output, bits 13 & 15 MUST always be high (1). The long hexidecimal value you see in the code lays out like this in binary (1's and 0's) if we were to set both pins 5 & 9 low.

```
HEXIDECIMAL    F    F    F    F    A    F    F    F
     BINARY 1111 1111 1111 1111 1010 1111 1111 1111
BIT NUMBERS 3322 2222 2222 1111 1111 1100 0000 0000
            1098 7654 3210 9876 5432 1098 7654 3210
```

The 'A' value occupies bits 12 - 15. Only these bits will be referred to from now on. The 'F' values (BINARY 1111) are used with the POTBITS mask sent to AllocPotBits( ) in the get_bits( ) routine and must be set this way for the WritePotGo( ) function to work. Again, please refer to the Amiga manuals for a complete explanation of other bit functions. Now, back to the out( ) function.

Three variables will be used, 'y' (the inactive bit state), 'z' (the commanded state) and 'polar' (detects polarity and is used to modify y and z). In the polarity( ) function, we set polar to 0 (bits 12 -15) for the 'U' command and to 5 for anything else ('D' is assumed if 'U' wasn't received.) Yet another switch statement is used to act on the commands 1, 2, or 3 in the out( ) function. Let's assume that 1 was received so pin 5 has to be pulsed. We will also assume that the last polarity (or the default) was 'U' so we will be pulsing pin 5 LOW. The inactive state (y) is always set to F (1111). Each case clause sets z to drive the commanded output low. Since we are assuming the output 1 command, z will become 1110. The zero is bit 12 (pin 5) to create the low pulse. Both direction bits (13 & 15) and bit 14 (pin 9) are set high. A message is printed announcing the event and the switch/case statement is broken out of. The polar variable is tested for 0101 on bits 12 -15. If the test is true, then we would invert y and z according to the value of polar. In this case polar = 0 so the inversion does not take place. 'z' is written to the port via WritePotGo( ) (an Amiga function), the Timer is started and we Wait( ) for the Timer signal to return. When the signal occurs, the message is retrieved (Amiga etiquette) and y is written to deactivate the pin(s). That was simple enough. But, what's this inversion stuff? I'm glad you asked.

I have referred to exclusive ORing previously in this article and it is the mechanism I use to invert. Some of you may not know how an exclusive OR (let's call it XOR from now on) works or even what it is. The XOR operation is really pretty simple. A normal OR operation takes two bits and if either OR both is a '1' then the result is '1'. An XOR operation will only produce a '1' if and only if ONE of the bits is a '1'. The other must be '0' (exclusive). If both are '0' then the result is '0'. If both are '1', the result is also '0' (inversion). Here is a chart (truth table) which illustrates the inversion process.

The first set shows low pulsing.
The second set shows high pulsing (inversion).
All values are binary (bits 15 - 12).

```
                INITIAL                    RESULT
COMMAND       y       z       polar      y       z
  GPO 1    1111    1110    0000       1111    1110
  GPO 2    1111    1011    0000       1111    1011
  GPO 3    1111    1010    0000       1111    1010

  GPO 1    1111    1110    0101       1010    1011
  GPO 2    1111    1011    0101       1010    1110
  GPO 3    1111    1010    0101       1010    1111

                                      P P     P P
                                      I I     I I
                                      N N     N N
                                      9 5     9 5
```

Notice, in the second set, that wherever polar has a 1 and y or z has a 1, the result is 0. Also notice that the direction bits stay high and the uncommanded pin stays in the inactive state (low for high pulsing). The big advantage to XOR is the avoidance of excessive 'if' testing. Once you understand XORing, it can save a fair amount of code. We didn't have to fool with this in the non-ARexx version (GPIO) since no polarity command is available.

This completes the out( ) function (thank goodness). All that remains is in( ) (command wise that is).

in( ) *Invoked by the GPI command* *Listing 4*

You may recall in GPIO that we waited on the trigger in the main( ) routine. GPIO_Rx Waits( ) in this routine in order to isolate the ARexx and Timer Wait( )s. Everything else is the same so I won't rehash it.

Remember, if a command match occurred when the ARexx message was processed, the ARexx message is not replied to until the command is processed. The script, therefore, is kept on hold. None of the commands do any busy-waiting (Amiga taboo). The Timer makes this possible when polling must be used. For those who do not know, the Amiga's Wait( ) function provides the means to keep a program in stasis until an event occurs. Non-multitasking machines can busy wait without ill effects but the Amiga's O.S. has more important things to do. A busy-wait loop just eats up valuable processor time and can even invoke a crash.

Termination

cleanexit( ) *Listing 4*

When GPIO_Rx receives the 'EXIT' command or a control-F, this routine is called. The only difference from GPIO is the ARexx port has to be closed via killRexxPort( ) in gpio_rexx.c.

→

killRexxPort( ) *Listing 5*

The ARexx port is first removed so we don't get any traffic during shutdown. Next, we see if a message has not been answered and if so, it is answered to make ARexx happy. The library base is closed, the port is deleted and our internal flag is set to 0 so other routines will know we don't have an ARexx port.

On To Other Subjects—Hardware

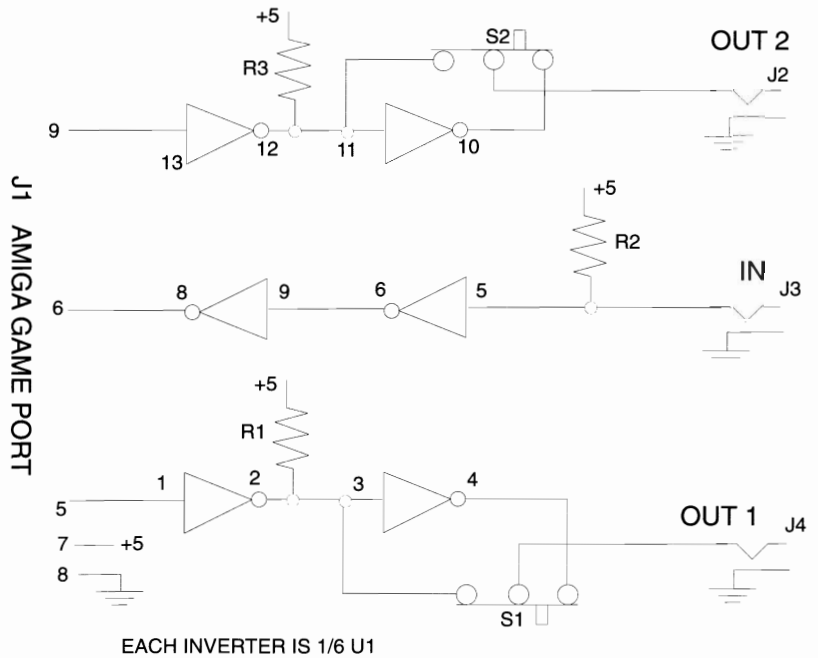
The software has been thoroughly thrashed to death, so let's look at the hardware. I wish I had a means to fabricate printed circuit boards but since I do not, I am not offering any kits or assembled units. The circuit is very simple, however, and isn't sensitive to noise so any construction technique can be used (please refer to figure one.)

It can't get much easier than this. Switches S1 and S2 are optional if the polarity command is all you need. All they do is route the appropriate inverter output, second inverter for HIGH-LOW-HIGH triggers and the first inverter for LOW-HIGH-LOW triggers (assuming GPIO_Rx is using the default polarity). The input buffering was done with two inverters since they were available on the same chip. No need to put more than necessary in the circuit. The 74C14 was chosen for its schmitt trigger feature. Schmitt triggers give a nice sharp transition even with a slow or noisy input. The game port outputs are not noisy but they do have fairly large capacitors that round off the transitions. All power is taken from the game port. The CMOS chip draws minuscule amounts. The resistors provide current taps for TTL compatibility. I used RCA jacks for all GPIO inputs and outputs since this seems to be the convention in video equipment. A DB 9 male to female cable is required to connect the GPIO module to your Amiga. Some vendors sell these as joystick extenders.

Everyone should be aware that some video equipment requires either a relay contact closure or an open collector transistor to drive their inputs. You will have to consult the documentation provided with your equipment for this information. Figure Two is a simple relay circuit you can build to give your outputs relay closures. *The game port will not supply enough current for this circuit* so a power source is up to you. Some video equipment have power taps for this purpose, check your docs.

The relay circuit would be triggered by the GPIO circuit. There are two modes of operation, pulsed and latched. In pulse mode the flip flop is bypassed. In latched mode two GPO pulses are required, one to set the flip flop and one to clear it. Choose the mode via XJ1. A single pole/double-throw switch could be substituted for XJ1. The momentary switch allows manual latch clearing if desired. The JK flip flop is configured as a 'T' or toggle type by connecting both J and K to 5 volts. One pulse sets it and the next pulse clears it. Always use the default or 'U' polarity to drive this module. Switch S2 in the relay circuit does any polarity switching needed.

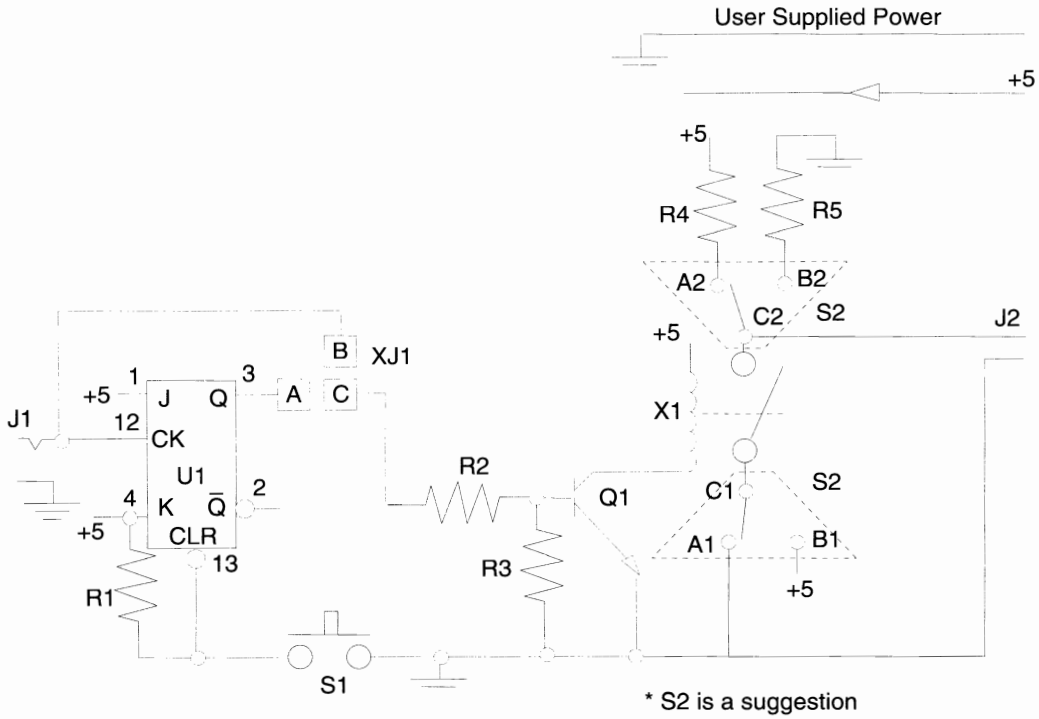
Figure One



PARTS LIST

U1	74C14
R1	3K
R2	3K
R3	3K
J1	DB 9 FEMALE
J2	RCA JACK
J3	RCA JACK
J4	RCA JACK
S1	SPDT SWITCH
S2	SPDT SWITCH
MISC	PERF BOARD
	SOCKET
	WIRE
	ENCLOSURE

Figure Two



PARTS LIST

U1	74107
J1	RCA JACK
J2	USER DEP.
R1	4.7K Ω
R2	1K Ω
R3	720 Ω
R4	3.3K Ω
R5	3.3K Ω
Q1	NPN - 2N3904 *or eqivalent*
S1	SPST MOMENTARY
S2	DPDT SLIDE SWITCHES
XJ1	3 HEADER PINS or SPDT SWITCH
X1	TTL RELAY
MISC	PERF BOARD SOCKET(S) WIRE ENCLOSURE

The GPIO input will work on most anything. The only exception will be equipment that uses open collector outputs which require a pull-up resistor in most cases. There are too many possibilities here to cover them all. I will be glad to help you out if all pertinent information is sent to me for analysis and you provide return postage.

Just to remind you, the input can be triggered with a standard Amiga joystick if that is all you need, and this is a good way to preview a script. No additional interface hardware will be needed; just plug in the joystick normally.

Speaking of scripts, I've included two examples. One uses the non-ARexx version (GPIO) (Listing 7). The second (Listing 8) controls the Video Toaster's Switcher.

GPIO Demo script *Listing 7*
Use the Amiga's execute command to run this.

This example does not call any graphic display programs since they would have to be provided. All it does is display the current directory with the standard 'DIR' command. The output commands were included for illustration but don't do any real work.

I have successfully written a script which plays animations with the SuperView utility and controls a Magni Genlock with one GPIO output. Timing can get a little tricky. You should generally use the 'run' command to launch graphic display programs followed immediately by a 'wait' command. The wait command can be adjusted so the genlock is triggered to fade the display out while the next graphic is loaded. An animation player with an ARexx port would be ideal, but then you would have to own ARexx to use it. Alas, the RUN/WAIT combination is the best solution for the non-ARexx GPIO version.

Now for the ARexx version....GPIO_Rx!!!

GPIO_Rx/Video Toaster script *Listing 8*
Use RX to run this (supplied with the ARexx package).

This demonstrates the use of several of the Toaster's ARexx commands. Refer to your Toaster documentation for further experimentation reference.

Note the use of subroutines for sending GPIO_Rx commands. This was done to save extra 'address' commands in the script and improve readability. Without the subroutines the 'address' commands would have to surround every GPIO_Rx command issued. ARexx has to know which port to send the command to. This script uses the Switcher by default so the "address GPIO.rxport" command must be used to tell ARexx

→

that GPIO_Rx is to receive the next command. The lone 'address' command causes ARexx to revert to the port address in use prior to the last issued 'address' command (The Switcher).

The ARexx support library was opened just for illustration and in case you want to use AREXX's DELAY( ) function (or any other for that matter). As in the GPIO Demo, the output, polarity and timebase commands are included as examples but do not do any real work.

Additional Info

The GPIO version is capable of being made resident so it does not need to be loaded each time it is executed (once per GPIO command.) If you are using the standard 1.3 shell, just make it resident with the RESIDENT command. If you are using the ARP shell, you will need to use the ARES command with the NOCHECK option since no matter how I write the code, it always would error out with a program checksum error. I have tested it extensively with the NOCHECK option and have had no problems. You can use the FORCE option also so it won't pause the first time GPIO is used.

When using GPIO_Rx, always include the 'EXIT' command so GPIO will terminate gracefully. For some reason the control-F exit does not always work. During script experimentation you are likely to have a few errors in your script which will prevent it from reaching your 'EXIT' command. Do not fear, GPIO_Rx will simply produce the "CAN'T GET THE BITS" error because it is already running. The script will still find the port and function correctly. Once the script is debugged, the 'EXIT' command will be reached and GPIO_Rx will terminate. BE SURE TO ENCLOSE GPIO_Rx's 'EXIT' command in single quotes or ARexx will think that it is its exit command and not send it to GPIO_RX . You can always rewrite GPIO_Rx.c to change the 'EXIT' command's syntax if you like.

I hope you find this as useful as I have. The uses for these programs is limited only by your imagination. GPIO_Rx can be used in any script which calls any other ARexx-equipped program as long as that program does not use the joystick port.

Any questions you have may be sent to me at:

Ken Hall
105 Maple Ct.
Ozark, AL 36360 *or in care of AC's TECH.*

When, writing directly to me, please include return postage for your reply and any reference materials you wish returned. Correspondence which adds to my programming knowledge will not require return postage unless the reference materials are bulky and you want them back.

Have fun and happy sequencing!!!

Listing One

```
/*         Listing 1         */
/* GPIO include and define declarations */

#include <libraries/dos.h>
#include <exec/types.h>
#include <exec/devices.h>
#include <devices/timer.h>
#include <exec/errors.h>
#include <exec/exec.h>
#include <stdio.h>

#ifdef MANX
#include <functions.h>
#endif

#ifdef LATTICE
#include <pragmas.h>
#endif

#define POS-MSG "GPIO: Waiting + Input"
#define NEG-MSG "GPIO: Waiting - Input"

#define OUT1-MSG "GPIO: Pulsed 1"
#define OUT2-MSG "GPIO: Pulsed 2"
#define OUT3-MSG "GPIO: Pulsed BOTH"

#define EXIT-MSG "GPIO: Exit."
#define BREAK-MSG "GPIO: Break exit."
#define TIME-PORT-ERR "GPIO ERROR: Can't get the timer port"
#define TIME-DEV-ERR "GPIO ERROR: Can't get the timer device"
#define NO-POTGO "GPIO ERROR: Can't get the POTGO resource"
#define NO-BITS "GPIO ERROR: Potbits in use"

#define UNLESS(x) if(!(x))
```

Listing Two

```
 /****************************************
 *               Listing 2              *
 *  INIT GPIO gameport and timer data     *
 * Satelite rouitnes for GPIO and GPIO-Rx *
 * (c) 1991 Ken Hall                      *
 ****************************************/

#include "gpio.h"

static struct timerequest Time-Req ;
struct MsgPort       *Timer-Port = NULL;
struct PotgoBase *PotgoBase;
ULONG  potbits=0L;
ULONG timebase=500L;

ULONG gpio-init();
void get-bits();
void send-timer();
void kill-bits();
void kill-time();

#define POTBITS 0xF000L

/*-INITIALIZE-----------------------*
 *          Allocate all resources                    *
  *--------------------------*/
ULONG gpio-init()
{

  UNLESS (Timer-Port = CreatePort("GPIO-Timer-Port", 0L))
    cleanexit(TIME-PORT-ERR,RETURN-FAIL);

  if (OpenDevice(TIMERNAME, UNIT-MICROHZ,
     (struct timerequest *) &Time-Req, 0L))
    cleanexit(TIME-DEV-ERR,RETURN-FAIL);
```

```
  Time-Req . tr-node . io-Message . mn-ReplyPort = Timer-Port ;
  Time-Req . tr-node . io-Command = TR-ADDREQUEST ;
  Time-Req . tr-node . io-Flags = 0L ;
  Time-Req . tr-node . io-Error = 0L ;

  get-bits();
  WritePotgo(0xffffffffL,potbits);
return(1L << Timer-Port -> mp-SigBit) ;
}

/*-TIMER-------------------------*
 *           Send out a timer request                    *
  *---------------------------*/
void send-timer()
{
Time-Req . tr-time . tv-secs = 0L ;
Time-Req . tr-time . tv-micro = timebase;
SendIO((char *) &Time-Req . tr-node) ;
}

/*-BITS--------------------------*
 *       Allocate the pot bits                           *
  *---------------------------*/
void get-bits()
{
  UNLESS (PotgoBase=(struct PotgoBase *)OpenResource("potgo.resource"))
    cleanexit(NO-POTGO,RETURN-FAIL);

  potbits=AllocPotBits(POTBITS);

  if(potbits!=(POTBITS))
    cleanexit(NO-BITS,RETURN-FAIL);

}

/*-BITS--------------------------*
 *           Free the pot bits                           *
  *---------------------------*/
void kill-bits()
{
  if(potbits != 0L)
    FreePotBits(potbits);
  potbits=0L;
}

/*-SHUT DOWN---------------------*
 *           Free all resources                          *
  *---------------------------*/
void kill-time()
{
  AbortIO((char *) &Time-Req . tr-node);
  if (Time-Req . tr-node . io-Message . mn-ReplyPort)
    CloseDevice(&Time-Req);
  if (Timer-Port)
    DeletePort(Timer-Port);
}
```

Listing Three

```
  /*                Listing 3                    */
 /*-------------------------*
 *                                                 *
*                                                  *
*   GPIO  - Run this task everytime to:            *
*                                                  *
*                            gpio p   Wait + level *
*   COMPILE: GPIO.c          gpio n   Wait - level *
*            GPIO-init.c     gpio 1   Output 1 pulse *
*                            gpio 2   Output 2 pulse *
*                            gpio 3   Both Outputs *
*                            gpio     info         *
*                                                  *
*                                                  *
*   LINK: gpio gpio-init c.lib                     *
*   small code and data. No special link needs.    *
 *                                                 *
  *================================================*/
```

```
#include "GPIO.h"

static long timerbit; /* The timer port signal bit    */
static BYTE GPI-flag=0,k;

extern struct MsgPort *Timer-Port;
extern ULONG time-default;

#define VERSION "GPIO version 1.2\n copyright 1991 by Ken Hall"

void out(),in(),cleanexit();

/*-MAIN----------------------*/
/*                  Main Program                    */
/*                                                  */
 /*         This is the main program loop.          */
  /************************************************/

main(argc,argv)
SHORT argc;
char **argv;
{
ULONG sig;
SHORT ImAlive=0;
char y='0';

extern ULONG potbits;
extern ULONG gpio-init();
extern char *gpio-errors[];

   if (argc>=2)
     y=toupper(*argv[1]);

   timerbit=gpio-init();

   switch(y){
     case '1':
     case '2':
     case '3':
       out(y);
       cleanexit(0L,RETURN-OK);
     case 'P':
     case 'N':
       in(y);
     break;
     default:
       printf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
         VERSION,
         "SYNTAX:",
         "GPIO P   Wait + level",
         "GPIO N   Wait - level",
         "GPIO 1   Output 1 pulse",
         "GPIO 2   Output 2 pulse",
         "GPIO 3   Both Outputs",
         "GPIO     This display",
         "Ctrl-F   abort GPIO wait");

       cleanexit(0L,RETURN-OK);
   }
   ImAlive=1;
   while( ImAlive ){
     sig=Wait( timerbit | SIGBREAKF-CTRL-F);

     if (sig == SIGBREAKF-CTRL-F){
       cleanexit(BREAK-MSG,RETURN-FAIL);
     }else
       GetMsg(Timer-Port);
       k=*(BYTE *)0xbfe001L;
       k&=0x80;

       if(k^GPI-flag)
         break;

       send-timer();
   }
   cleanexit(0L,RETURN-OK);

}
```

```
/*-GPIO-----------------------*
 *            Handle GPI output requests            *
  *-------------------------*/
void out(p)
char p ;
{
ULONG y,z,savebase;

extern ULONG potbits,timebase;

     y=0xffffffffL;
  switch (p){
    case '1':
      z=0xfffefffL;
      printf("%s\n",OUT1-MSG);
    break;
    case '2':
      z=0xfffbfffL;
      printf("%s\n",OUT2-MSG);
    break;
    case '3':
      z=0xfffafffL;
      printf("%s\n",OUT3-MSG);
    break;
    default:
      return;
  }

    timebase=500L;

    WritePotgo(z,potbits);
    send-timer();

    Wait(timerbit);
    GetMsg(Timer-Port);

    WritePotgo(y,potbits);
    WritePotgo(y,potbits);
}

/*-GPIO-----------------------*
 *             Handle GPI input requests             *
  *-------------------------*/
void in(p)
char p ;
{
long sig;
char *mm;

  switch (p){
    case 'P':
      mm=POS-MSG;
      GPI-flag=0x00;
    break;
    case 'N':
      mm=NEG-MSG;
      GPI-flag=0x80;
    break;
    default :
      cleanexit(0L,RETURN-OK);
  }
  send-timer();
  printf("%s\n",mm);
}

/*-SHUT DOWN-----------------*
 *              Close up Shop               *
  *--------------------*/
void cleanexit(s,n)
UBYTE *s;
LONG n;
{
  if (s)
    printf("%s\n",s);
  kill-time();
  kill-bits();
  exit(n);
}
```

Listing Four

```
  /*                 Listing 4                    */
 /*-----------------------*
 *                                                 *
*                                                   *
*   GPIO-Rx - Run this task and use ARexx(tm) to    *
*             speak to it.                          *
*       (c) 1991 Ken Hall                           *
*              AREXX SYNTAX:  address "GPIO.rxport"  *
*                             gpi p   Wait + edge    *
*   COMPILE: GPIO-Rx.c        gpi n   Wait - level   *
*            GPIO-init.c      gpo 1   Output 1 pulse *
*            GPIO-rexx.c      gpo 2   Output 2 pulse *
*                             gpo 3   Both Outputs   *
*    define: COMPLAIN=1/0     version info           *
*                             polarity up/down       *
*                             timebase 1-9 (*4167)   *
*                             exit    quit           *
*                                                    *
*   LINK: gpio-rx gpio-init gpio-rexx rexxglue       *
*   small code and data. No special link needs.      *
 *                                                  *
  *================================================*/

#include "gpio.h"
#include "gpio-rexx.h"

#define VERSION "GPIO-Rx version 3.0\n (c) 1991 by Ken Hall"

static ULONG timerbit;
static SHORT ImAlive = TRUE;
static ULONG polar=0xffff0fffL;
void rexx-host();
void cleanexit();

extern struct MsgPort *Timer-Port;

/* GPIO's functions  */
 void  in(),
       out(),
 set-time(),
 polarity(),
  version(),
gpio-exit();

    /**************************
    * GPIO's command list.     *
    * MUST be NULL terminated  *
    **************************/

struct rexxCommandList RxCmdList[] = {
   { "GPI",      (APTR)&in },
   { "GPO",      (APTR)&out },
   { "EXIT",     (APTR)&gpio-exit },
   { "VERSION",  (APTR)&version },
   { "POLARITY",(APTR)&polarity },
   { "TIMEBASE",(APTR)&set-time },
   { NULL };

/*-MAIN-----------------------*/
/*                  Main Program                   */
/*                                                 */
 /*          This is the main program loop.        */
  /************************************************/

main()
{
ULONG sig;
ULONG rexxbit;
ULONG class;

extern ULONG gpio-init();
extern ULONG callRexxPort();
extern void answerRexxPort();

   timerbit=gpio-init();
```

```
    UNLESS (rexxbit = callRexxPort("GPIO.rxport",RxCmdList, &rexx-host))
      cleanexit("GPIO ERROR: NO ARexx PORT\n",RETURN-FAIL);

    version();

    while( ImAlive ){
      sig=Wait(SIGBREAKF-CTRL-F | rexxbit );

      if(sig & rexxbit){
        answerRexxPort();
      }else{
        if (sig & SIGBREAKF-CTRL-F)
          cleanexit(BREAK-MSG,RETURN-FAIL);
      }
    }
   /* Get here via ARexx and the GPIO 'exit' command ONLY */
    cleanexit(EXIT-MSG,RETURN-OK);

}

/*-REXX-------------------------*
 *        This is our main Rexx->GPIO handler.           *
 *        Calls the function corresponding to the command *
 *        Waits for it to return.                        *
  *--------------------------*/
void rexx-host(msg, dat, p)
register struct RexxMsg *msg ;
register struct rexxCommandList *dat ;
char *p ;
{
char *pp=p;
p++;
  while(*p++) /* Parse out the last character argument */
  pp++;
  ((SHORT (*)())(dat->function-address))(msg, pp) ;
  *pp=0;
}

/*-GPIO-------------------------*
 *             This handler sets the exit flag.          *
  *--------------------------*/
void gpio-exit(msg, p)
struct RexxMsg *msg ;
char *p ;
{
   ImAlive = 0 ;
}

/*-GPIO-------------------------*
 *       This handler prints the version of the program.  *
  *--------------------------*/
void version(msg, p)
struct RexxMsg *msg ;
char *p ;
{
printf("\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n\n",
         VERSION,
         "AREXX SYNTAX:  address 'GPIO.rxport'",
         "GPI P      Wait + level",
         "GPI N      Wait - level",
         "GPO 1      Output 1 pulse",
         "GPO 2      Output 2 pulse",
         "GPO 3      Both Outputs",
         "POLARITY U HIGH-LOW-HIGH for GPO commands",
         "POLARITY D LOW-HIGH-LOW for GPO commands",
         "TIMEBASE n Sample input n*4167 micro-sec",
         "VERSION    This info",
         "EXIT       Quit GPIO",
         "Ctrl-F     Abort GPIO");
}
/*-GPIO-------------------------*
 *       Set the output polarity (pulse up or down)      *
 *                                                       *
 *  Test command, assign mask bits, set port, tell user  *
  *--------------------------*/
void polarity(msg,p)
struct RexxMsg *msg;
char *p;
{
```



```
extern ULONG potbits;

  if (*p=='U' || *p=='H'){
    polar = 0xffff0fffL;
    WritePotgo(0xffffffffL,potbits);
    printf("%s\n","GPIO: Polarity HIGH-LOW-HIGH");
  }else{
    polar = 0xffff5fff;    /* Signal inversion */
    WritePotgo(0xffffafffL,potbits);
    printf("%s\n","GPIO: Polarity LOW-HIGH-LOW");
  }
}

/*-GPIO-------------------------*
 *       Set the time base for GPI input check           *
  *--------------------------*/
void set-time(msg,p)
struct RexxMsg *msg;
char *p;
{
ULONG t=1;

extern ULONG timebase;

  if (*p)
   t=(ULONG)(*p & 0x0F);        /* Mask out upper nibble */

  timebase=(t>0) ?t * 500L :500L;  /* 0 is taboo */

  printf("GPIO: TimeBase set to %ld microseconds\n",timebase);
}
```

→

```
/*-GPIO-------------------------*
 *            Handle GPI output requests              *
  *-------------------------*/
void out(msg,p)
struct RexxMsg *msg ;
char *p ;
{
ULONG y,z,savebase;
extern ULONG potbits;

  y=0xffffffffL;

  switch (*p){
    case '1':
      z=0xfffefffL;
      printf("%s\n",OUT1-MSG);
    break;
    case '2':
      z=0xffffbfffL;
      printf("%s\n",OUT2-MSG);
    break;
    case '3':
      z=0xffffafffL;
      printf("%s\n",OUT3-MSG);
    break;
    default:
      return;
  }

  if(polar == 0xffff5fffL){ /* Test for inversion need */
puts("invert");
    y^=polar;          /* Invert */
    z^=polar;
  }

    savebase=timebase;
    timebase=500L;

    WritePotgo(z,potbits);
    send-timer();

    Wait(timerbit);
    GetMsg(Timer-Port);

    WritePotgo(y,potbits);
    WritePotgo(y,potbits);
    timebase=savebase;
}

/*-GPIO-----------------------*
 *             Handle GPI input requests            *
  *-------------------------*/
void in(msg,p)
struct RexxMsg *msg ;
char *p ;
{
ULONG sig;
char *mm;
BYTE level,bitis;

  switch (*p){
    case 'P':      /* Wait for Positive (high) level */
      mm=POS-MSG;
      level=0x80;
    break;
    case 'N':
    default :      /* Default to Negative (low) level */
      mm=NEG-MSG;
      level=0x00;
  }

  printf("%s\n",mm);
  send-timer();    /* We only use the timer when needed */
  while(1){
    sig=Wait(timerbit | SIGBREAKF-CTRL-F);
    if (sig & SIGBREAKF-CTRL-F){
      cleanexit(BREAK-MSG,RETURN-FAIL);
    }

    GetMsg(Timer-Port);

    bitis=*(BYTE *)0xbfe001L; /* Read port */
    bitis &=0x80;             /* Mask unused bits */

    if(bitis==level)          /* Test */
      break;                  /* Quit loop if equal */

    send-timer();  /* Not yet - restart the timer */
  }
}

/*-SHUT DOWN-----------------*
 *              Close up Shop               *
  *--------------------*/
void cleanexit(s,n)
UBYTE *s;
LONG n;
{
  if (s)
    printf("%s\n",s);
  kill-bits();
  kill-time();
  killRexxPort();
  exit(n);
}
```

Listing Five

```
/*               Listing 5               */
/*****************************************
*      Satelite functions for GPIO-Rx      *
*      (c) 1991 Ken Hall                   *
 *****************************************/

#include "gpio-rexx.h"

#define UNLESS(x) if(!(x))

/* our rexx port */
static struct MsgPort *rexxPort ;

/* our command list */
static struct rexxCommandList *globalRxCmdList ;

/* signal bit to wait on for Rexx */
static ULONG  rexxPortBit ;

/* host routine function pointer  */
static void   (*host)() ;

/* outstanding Rexx message */
static struct RexxMsg *pendingRexxMsg ;

struct RxsLib *RexxSysBase ;

ULONG callRexxPort();
void  answerRexxPort();
char  check-command();
void  killRexxPort();

/*------------------------*
 *                                          *
 *   This is the rexx port startup routine.       *
 *   It is used to establish the rexx port        *
 *   and returns the signal bit or                *
 *   0 if the port can't be granted.              *
  *                                         *
   *----------------------*/

ULONG callRexxPort(RxPortName, RxCmdList, UsrFnctPntr)
char *RxPortName ;
struct rexxCommandList *RxCmdList ;
void  (*UsrFnctPntr)() ;
{
   struct MsgPort *FindPort() ;
```

```
   struct MsgPort *CreatePort() ;

   if (RxCmdList == 0L || UsrFnctPntr == 0L)
      return(0L) ;

   if (rexxPort == 0L) {
      Forbid() ;
      UNLESS (FindPort(RxPortName))
         rexxPort = CreatePort(RxPortName, 0L) ;
      Permit() ;
      if (rexxPort != 0L)
         rexxPortBit = 1L << rexxPort->mp-SigBit ;
   }
   globalRxCmdList = RxCmdList ;
   host = UsrFnctPntr ;
   return(rexxPortBit) ;
}
/*-------------------------*
 * Here we handle any REXX message.                    *
 * This will process multiple command messages and     *
 * only answers. No Macro cabability, No communication *
 * with another host application.                      *
 * If a command is not understood we just print some   *
 * text for user info. ARexx will ALWAYS think         *
 * everything is perfect until we get an ARexx 'exit'  *
 * command or the host port vanishes.                  *
 * You can #define the COMPLAIN directive to tell      *
 * ARexx if the command did not match.                 *
*                                                      *
  *                                                    *
   *------------------------*/
void answerRexxPort()
{
   register struct RexxMsg *GetMsg() ;
   register struct RexxMsg *RexxMsg ;
   register struct rexxCommandList *RxCmdList ;
   register char *ArgParsePntr = 0L;

   if (rexxPort == 0L)
      return ;

   while (RexxMsg = (struct RexxMsg *)GetMsg(rexxPort)) {

     ArgParsePntr = (char *)RexxMsg->rm-Args[0] ;
     while (*ArgParsePntr > 0 && *ArgParsePntr <= ' ')
       ArgParsePntr++ ;
       RexxMsg->rm-Result1 = 0 ;  /* ALWAYS happy */
       RexxMsg->rm-Result2 = 0 ;

       /* Look for a command match */
       pendingRexxMsg = RexxMsg;
       for (RxCmdList = globalRxCmdList;
            RxCmdList->function-name;
            RxCmdList++) {
        if (check-command(RxCmdList->function-name,
            ArgParsePntr) == 0) {
             host(RexxMsg,RxCmdList,
               ArgParsePntr+strlen(RxCmdList->function-name));
             break ;
        }
       }
       if(RxCmdList->function-name==0L){
         printf("GPIO ERROR: Unknown command ->%s\n",
              (char *)RexxMsg->rm-Args[0]);
#ifdef COMPLAIN
         RexxMsg->rm-Result1 = RXERRORNOCMD;
#endif
       }
       if (pendingRexxMsg){
         ReplyMsg(RexxMsg) ;
         pendingRexxMsg = 0L ;
       }
   }
}
```



```
/*-------------------------*
 *   This is the function we use to see if the command *
 *   matches the command string.  Not case sensitive.  *
*
 *   Returns the last character read from the command  *
 *   list name. If we got a match then c will be the   *
 *   NULL terminator                                   *
  *------------------------*/
char check-command(c, m)
register char *c, *m ;
{
   while (*c && (*c == toupper(*m))){
     c++ ;
     m++ ;
   }
   return(*c) ;
}

/*-------------------------*
 *   This function closes down the Rexx port.          *
 *   It removes the Rexx port,                         *
 *   replies to an outstanding message.                *
 *   Closes the Rexx library.                          *
 *   Zaps the port and signal.                         *
  *------------------------*/
void killRexxPort()
{
   if (rexxPort) {
      RemPort(rexxPort) ;

      if (pendingRexxMsg) {
         pendingRexxMsg->rm-Result1 = RXERROREXIT ;
         ReplyMsg(pendingRexxMsg) ;
```

➔

```
        pendingRexxMsg = 0L ;
      }
      if (RexxSysBase) {
        CloseLibrary(RexxSysBase) ;
        RexxSysBase = 0L ;
      }
      DeletePort(rexxPort) ;
      rexxPort = 0L ;
   }
   rexxPortBit = 0 ;
}
```

Listing Six

```
 /*        Lising 6             */
/*---------------*
  *   Include for gpio-rexx.c   *
   *-------------*/
#include <rexx/rxslib.h>
#define RXERROREXIT (100)
#define RXERRORNOCMD (30)

/*---------------*
 *   CommandList Structure       *
  *--------------*/
struct rexxCommandList {
   char *function-name ;
   APTR function-address ;
} ;
```

Listing Seven

```
; GPIO (non-AREXX DEMO
; Use the Amiga's Execute command to run this

; make GPIO resident. Uncomment the appropriate lines
; depending on which shell you are running
; If no shell then residency is not possible but the
; script will work ok anyway

; For the Amiga 1.3 shell
;resident GPIO remove
;resident GPIO

; For the ARP shell
ares GPIO remove
ares /Software/GPIO NOCHECK FORCE

gpio n
dir

gpio 1
gpio 2
gpio 3
```

Listing Eight

```
/* gpi.rexx */
/* Control the Switcher using Arexx and GPIO */
/* AREXX version 1.5 used */

say
say "**********************"
say "**  GPIO-Rx Demo  **"
say "**********************"


if ~show('L',"rexxsupport.library") then do
   if addlib('rexxsupport.library',0,-30,0) then
      say 'added rexxsupport.library'
   else do;
      say 'support library not available'
      exit 10
   end
end
```

```
TRUE = 1
FALSE = 0

Switcher = 'ToasterARexx.port'
addlib(Switcher,0)
address command
RUN GPIO-RX
call delay(50)
/* if the Switcher is not around, start it */

if show(port,Switcher) = FALSE
then
  do
  say "Running the switcher - Please stand by"
  'Run ARexxWait'
  'Run Switcher'
  address 'ARexxWait.port' 'wait'
  end

address                                  /* disable DOS commands */

/*- KEEP EVEYTHING FROM HERE UP THE SAME..NO CHANGES--*/

Switcher(TOSW)
Switcher(TOWB) /* Always have a switcher command first */
say
say "Connect a joystick to gameport 1 (Next to the mouse)"
say " If the Toaster CG is not currently loaded then do"
say "  it now. Press the control key twice then the"
say "  left alt key twice to go to the switcher. Repeat"
say "  to redisplay the CLI....Then"
say "Press the fire button "
call gpi n

/* The next six GPIO-Rx commands are for illustration  */

call gpo 1
call TIME 1
call gpo 2
call pol d
call gpo 3
call pol u

Switcher(KOFF)
Switcher(P001)
Switcher(M001)

/* Set up the first effect */
Switcher(FMLD,000)
Switcher(Grid,D23)
Switcher(TOSW) /* Use the 'TOSW' then 'TOWB' */
Switcher(TOWB) /* Commands in pairs to see the CLI */
say "Press the joystick's fire button"
call GPI n
Switcher(AUTO)


/* Set up the second effect */
Switcher(FMLD,001)
Switcher(Grid,A43)
Switcher(TOSW)
Switcher(TOWB)
say "Press the joystick's fire button"
call GPI n
Switcher(AUTO)

        /* IMPORTANT */
/* The next three switcher sequences access the CG */
/* You MUST activate the CG from the switcher manually */
/* No AREXX command was provided to do that from a script */
/* You can get from the switcher to the CLI and back by */
/* pressing the control key twice then the left alt key twice */

Switcher(BKLD,000)
Switcher(PAGE,001)
Switcher(GRID,B44)
Switcher(TOSW)
Switcher(TOWB)
say "Press the joystick's fire button"
call GPI n
```

```
Switcher(AUTO)

Switcher(PAGE,002)
Switcher(Grid,A35)
Switcher(TOSW)
Switcher(TOWB)
say "Press the joystick's fire button"
call GPI n
Switcher(AUTO)

/* Use the keyer with the last two CG pages */
Switcher(ODV1)
Switcher(KBLK)
Switcher(TOSW)
Switcher(TOWB)
say "Press the joystick's fire button"
call GPI n
Switcher(AUTO)

Switcher(TOSW)
Switcher(TOWB)

call GPE

say "This concludes the GPIO-Rx Demo"
say "   P.S. The switcher is still running "
say "        Use cntl cntl alt alt to go there"
exit

/*****************************************
* The following subroutines send specific *
* GPIO-Rx commands. If subroutines are not*
* used, the address statements will be in *
* primary script surrounding every GPIO-Rx*
* command.                                *
*****************************************/

GPV:
  address 'GPIO.rxport'
  VERSION
  address
return

GPE:
  address 'GPIO.rxport'
  'EXIT' /*BE SURE TO ENCLOSE THIS COMMAND IN SINGLE QUOTES*/
  address
return

GPI:
arg m
  address 'GPIO.rxport'
  gpI m
  address
return

GPO:
arg m
  address 'GPIO.rxport'
  gpO m
  address
return

POL:
arg m
  address 'GPIO.rxport'
  'POLARITY' m
  address switcher
return

TIME:
arg m
  address 'GPIO.rxport'
  TIMEbase m
  address
return
```

PROGRAMMING WITH THE

AREXXDB RECORDS MANAGER

by Benton Jackson
JMH Software of Minnesota Inc.

By now, most Amiga users have heard about *ARexx*, and its value to the Amiga. ARexx is an interpreted macro programming language developed by Bill Hawes. On the Amiga, its most important use is as a standard macro language. ARexx programs are thus called macros or scripts. An application that supports ARexx is called a host, and can be controlled by ARexx scripts. See "Interfacing Assembly Language Applications to ARexx," *AC's TECH*, Volume 1 number 2.

Furthermore, scripts can control more than one host at a time, effectively integrating the applications. One could write ARexx scripts that download your messages from a bulletin board using a terminal program, then show them in your editor. Next you could type in replies to these messages, and have another script post the messages back on the bulletin board.

Some applications for ARexx need a database. For example, the bulletin board scripts mentioned above could store old messages in a database. Many of the database managers on the Amiga have ARexx support. Unfortunately, the ARexx interfaces on all of them are oriented towards the user interface, and ARexx is usually used to bring in data from outside. What is needed is a database that is oriented towards serving ARexx. With this, the application program's user interface can be in control, rather than the database program's user interface.

Thus, the need for the ARexxDB Records Manager, from JMH Software of Minnesota Inc. This is a database manager dedicated entirely to ARexx scripts, or any other programs that can communicate with an ARexx host. RexxDB is the ARexx host that is included with the ARexxDB Records Manager product. RexxDB is designed to handle large and complex databases, using ARexx scripts to do any customization. It uses Linear Hashing to keep record storage and retrieval fast. BTree indexing is included to optimize searching for records. With RexxDB, ARexx scripts can retrieve records without anything appearing on the screen except a message on the status window. The status window can even be turned off if needed.

What can you do with RexxDB? It can do the normal mundane database chores such as address lists or inventory control, but really shines when used with other applications. With RexxDB, any Amiga application that supports ARexx now has the ability to keep records. For example, *AmigaVision* or *CanDo* can be used to enter data, and RexxDB could then store it. Or a database of actions to be performed on the Video Toaster could be stored by RexxDB. Parts data could be stored from a CAD package that supports ARexx, such as *IntroCAD Plus* or *ProVector*. Scripts are included with RexxDB to read and write database records directly from some of the popular ARexx compatible editors.

To illustrate, I'll create a phonebook/autodialer for VLT, called "VLTPhones." A phonebook/autodialer stores the phone numbers of bulletin boards. At the push of a button, the user chooses a phone number, and the autodialer tells the modem to dial the phone. VLTPhones uses RexxDB to store the phone numbers. The names of the bulletin boards are indexed to allow quick searching.

The complete script and RexxDB file are on the AC's TECH disk. VLT (A Valiant Little Terminal) is a freeware terminal program written by Willy Langeveld of Stanford Linear Accelerator Center (SLAC). VLTPhones uses rexxarplib.library, also by Willy Langeveld, for its windows and gadgets. To run VLTPhones, you will need the ARexxDB Records Manager, ARexx, VLT, rexxarplib.library, screenshare.library, and arp.library. VLT and the libraries are available on many bulletin boards, and also on the Fred Fish disks. ARexx is available from Bill Hawes, and ARexxDB is available from JMH Software of Minnesota Inc.

Before I get into the details of VLTPhones, a quick overview of RexxDB is in order. See the sidebar titled "RexxDB Function Summary." RexxDB is the foundation for a complete new multitasking database environment. The customizing database language is ARexx. The sourcecode editor can be your favorite ARexx compatible text editor, such as TxEd, CygnusEd Pro, or TurboText. You can use any program that supports ARexx as a user interface to the data, or you can

Harness the power of a database manager dedicated entirely to ARexx scripts, or any other programs that can communicate with an ARexx host.

design your own user interface in ARexx. If any new and/or improved ARexx supporting product comes along, you can fit it into your environment. Now programmers and developers can be released from the limitations of the traditional "integrated" database environments.

RexxDB files consist of two parts: the data part and the support part. All of the records in the data part are usually the same format. Records in the support part can be anything that supports the data, such as function scripts, BTree indexes, data patterns, or window templates. The files are segmented into data blocks, called "buckets." Each bucket can hold multiple records, or a long record can span multiple buckets. The Linear Hashing algorithm is used to distribute records into buckets, and add buckets when the file grows. The user can optimize each file for either space or speed, by setting how full the buckets should be kept, and by setting the bucket size.

Buckets are cached in memory—the first time a bucket is needed, it is read from the hard disk, and kept in memory. If it is needed again, it can be read from memory. You can set a maximum amount of memory to be used by caching. RexxDB runs a startup script when it loads. This script can run in the background, and watch for buckets that have changed and need to be written to the disk.

Access to RexxDB files is regulated by cursors. A cursor is the same as a "current record pointer." It also holds the AmigaDOS file handlers. When you use the OPENCURSOR function, you get an identifier for the cursor called a cursor number. You use this cursor number whenever you do something with the file. RexxDB is designed to work with multitasking. If many tasks want to access the same file, each task can open its own cursor.

Every record in a RexxDB file has a unique ID. Some database systems call this a "primary key." This is the string used to identify the record, and is also used to locate it in the file. For VLTPhones, we'll assign a number to each record. We could use the phone number as the ID, but that would prevent us from having duplicate phone numbers in the database, with different names. Here's an example of getting a cursor to the "VLTPhones" file, and using the cursor to read the record with the ID "1":

```
curs=OPENCURSOR("VLTPhones")
if curs="" then do
  say "Can't open VLTPhones file. Errornum="ERRORNUM
  exit
end
record=READREC(curs,"1")
call CLOSECURSOR(curs)
```

Notice that OPENCURSOR returns a NIL value ("") if the file isn't found, or any other error prevented the file from opening. When an error occurs, RexxDB sets the ARexx variable "ERRORNUM" to a number that identifies the error.

A record can be any ARexx string. Since ARexx strings are limited to 64K, records are limited to 64K. RexxDB imposes no restrictions on the format of a record. RexxDB does provide functions to facilitate records delimited into fields by a "field mark," ASCII 254. One of these functions, called "SEPARATE," converts a delimited record into an ARexx stem variable. For example, a VLTPhones record consists of four fields: "NAME," "PHONE," "PARITY," and "STRIP." To use SEPARATE, we build a pattern string, as follows (I'll print the field mark as "/"):

```
pattern="NAME/PHONE/PARITY/STRIP"
rec="JMH Software/424-5464/8N1/Y"
call SEPARATE("rec","/",pattern)
```

The SEPARATE function then creates these compound variables:

```
rec.NAME="JMH Software"
rec.PHONE="424-5464"
rec.PARITY="8N1"
rec.STRIP="Y"
```

Records can also be converted to stems without using a pattern. This is useful when you don't know ahead of time how many fields there are, such as with a list. Without a

→

pattern, the field names used are numbers, and the actual number of fields is stored in the variable ending in ".0". It also allows you to use a "do" loop, such as in this example that lists all the fields in a record:

```
call SEPARATE("rec","/")
do i=1 to rec.0
say rec.i
end
```

Indexing is used to locate a record using a key other than the ID, or to sort records. For example, if a file of people used the social security number as the record ID, then we can search for a person by name quickly if the name field is indexed. An index is a sorted list of key-ID pairs, or sometimes just a list of IDs. A BTree index is an index structured for fast storage and retrieval. In the VLTPhones file, each word in the NAME field is indexed, so that we can search for phone records by name or by part of a name.

Many RexxDB functions can be substituted by ARexx scripts for each file. If a record in the support part of a file has the same name as a RexxDB function, the record will be called as an ARexx script instead of the corresponding RexxDB function. This script is called a "function script." You can bypass a function script by adding a "DO" to the front of the function name. This is usually used in a function script to continue the function normally, without causing an infinite loop. Function scripts are frequently used to maintain indexes. Since function scripts are ARexx programs, any ARexx command can be used, not just RexxDB functions. You can even call other ARexx hosts! The applications are endless: transaction logging, security, virtual records (computed from other data), simulated formats (such as dBase), even a distributed database. Since the scripts are kept with the file, the code doesn't have to be repeated for every application that uses the file. Here's an example WRITEREC script that puts the record being written in the message line on the status window, and then writes the record normally:

```
/* WRITEREC- copy the record to the status window, then writes it
            normally. */
parse arg cursor,ID,rec

call POSTMESSAGE(rec)
return DOWRITEREC(cursor,ID,rec)
```

Notice that the program passes the same parameters as the functrion script to the function, and returns the same results.

So how do these function scripts get into the file? The easiest way is to use macros that work with an ARexx compatible editor. The macro needed to store a function script should copy the current editor file into an ARexx string, open a cursor to the support part of the file, and write the string as a record. You'll also need a macro to read the function script back into the editor. These macros are provided for some of the popular ARexx compatible editors with the ARexxDB Records Manager product.

The VLTPhones file has three function scripts: WRITEREC, DELETEREC, and CLEARFILE. The WRITEREC script calls the WRITEKEY function on every name in the NAMES field. The DELETEREC script calls DELETEKEY. And the CLEARFILE script calls the CLEARINDEX function:

```
/* CLEARFILE - Clears the NAMES index and the quick index. */

parse arg cursor

call DOWRITEREC(cursor,"%RECORDS%","") /* write an empty list. */
C=OPENINDEX(cursor,"NAMES")            /* open the index */
call CLEARINDEX(C)                     /* clear it */
call WRITEREC(C,"IDCTR","1")           /* re-set ID counter to 1.
                                          Note that you
                                          can use an index cursor to write
                                          to support part. */
call CLOSECURSOR(C)                    /* and close the index. */

return DOCLEARFILE(cursor) /* finish up by clearing the data part. */
```

These functions also maintain a "quick index," which is simply a sorted list of the record IDs (as fields in a record), using the LOCATEFIELDBY function. By the way, all of the functions that are called in this script with the "call" command return a "1" to indicate success, or a "0" to indicate failure. This could have been used for error checking, but to keep things simple I left that off.

An important feature needed on a multitasking database is record locking. This is used to prevent multiple tasks from working on the same record at the same time. Actually, RexxDB locks the record ID, instead of the record. This locks a

record that is about to be created. The VLTPhones script demonstrates this by locking any record that it reads or creates. If it locks a record, it sets the variable "locked" to the ID that it locked, to remember which ID is locked.

In VLTPhones, we use record locking when creating new ID's to determine if it's already created, or already in use. We store the counter for the next ID in the record "IDCTR" in the support part of the file. If we can't lock this ID, or the record already exists, we add 1 and try again. Here's the subroutine that does it:

```
newid:
  ID=readrec(scursor,"IDCTR")     /* read the counter. */
  if ID="" then ID=1              /* if not there, start with 1. */
  rec="X"                         /* dummy value to get loop going. */
  do while rec~=""                /* "rec" is NIL when record doesn't exist.*/
    if lockrec(cursor,ID) then do /* try to lock it */
      rec=readrec(cursor,ID)      /* if locked, try to read it. */
      if rec~="" then call unlockrec(cursor,ID) /* don't use if there. */
    end
    else rec="X"                  /* if can't lock, pretend it exists to keep going.*/
    if rec~="" then ID=ID+1       /* if still going, use next ID */
  end
  nullctr=ID                  /* remember we created ID, for updating IDCTR later.*/
  locked=ID                   /* remember we locked it, so we can unlock it. */
return
```

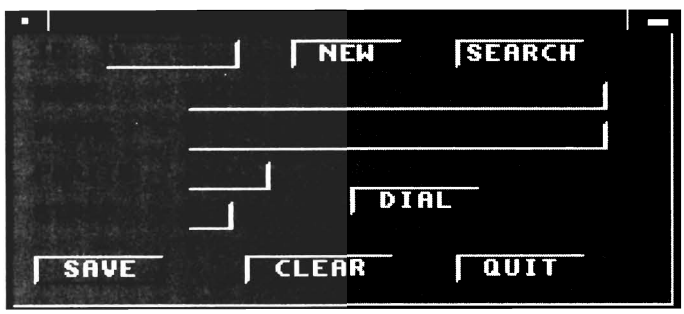


Figure One—The VLTPhones Entry Window

Now we're ready to talk about the entry window (Figure 1). We'll need a string gadget for the record ID, and a string gadget for each of the ID's. We'll also need various buttons for each of the actions that can be performed. The "NEW" button creates a new ID, as mentioned above. "SAVE" saves the current record. "CLEAR" clears out the current record, starting with blank fields. "DIAL" saves the record, dials the phone number in the record, and quits the window. "QUIT" does the same thing as the close gadget. Finally, the feature that most phone dialers don't have, and the reason to use a database: the "SEARCH" button. This is for searching for phone numbers by name, or any word in the name. Pressing the "SEARCH" button brings up the search window.

The search window starts out with one string gadget, for entering the name to search for. After the user enters a string in here, the NAMES index is searched for all IDs that were

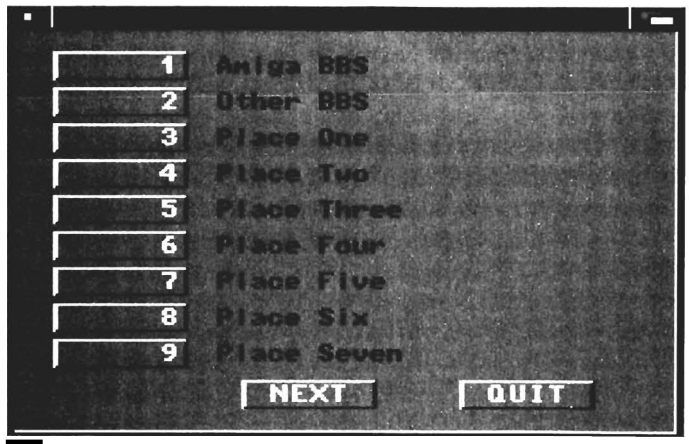


Figure Two—The VLTPhones Search Chooser

indexed with all the words in the name. If only one ID was found, then that ID is returned to the ID field of the main window, and the appropriate record read. If more than one ID is found, then a list is put up for the user to choose from, as shown in Figure 2.

The search algorithm used by this window illustrates the use of BTree indexing very nicely. Note that for an ID to come out of this, it must have been indexed with ALL the words that appear in the name. Here's a pseudo-code version of the algorithm (the ARexx implementation is part of the VLTPhones script):

```
if NAMES="" then
  use all ID's in the file.
else
  IDS=""
  for every NAME in NAMES
  if this is the first NAME, then
    read all ID's matching NAME into IDS
  else
    read all ID's matching NAME, and keep those that
    are already in IDS.
```

VLTPhones can be easily expanded to include more details about each phone number. The script calculates the position of the fields on the bottom, based on the number of fields. It also calculates the position of each field on the window. So to add fields, simply add the field name to the "pattern" variable, add 1 to the number of fields, and put the name of the field and the length into the stem variable at the top of the script. You might also need to add some commands to VLT to pass these new details on before dialing the phone.

To install VLTPhones on your system, first make sure VLT, RexxArpLib, and RexxDB are installed. Then, copy the "VLTPhones.vlt" script to your REXX: directory, and copy "VLTPhones.DB", "VLTPhones.DBO", "VLTPhones.SP", and "VLTPhones.SPO" to your RexxDB directory. (The scripts

➔

Say ARexx! Say ARexx! Say ARexx!

RexxDB Function Summary

Global functions

	Return version number or string.
	Return serial number.
GETMEMORY(newsize)	Set maximum RAM for buckets buffer.
	Write an unwritten bucket to disk.
	Write all unwritten buckets to disk.
	Get string of status values
	Displays message from script.
	Turns status window on or off.
	Return message about error number.
	Close all cursors.
	Prepare to shut computer off.

File functions

	Create a DB file.
	Obtain cursor to a file.
	Release file cursor.
	Dispose of a DB file.
	Clear data part of DB file.
SETFILE(cursor,min,max)	Set utilization factors for a file.
CURSORSTATUS(cursor[,num])	Get cursor status values.
FIRSTSCRIPT(cursor)	Get ID of first script for a file.
NEXTSCRIPT(cursor)	Get ID of next script for a file.
SETERROR(cursor,number)	Sets the error number for a cursor.

Record functions

LOCKREC(cursor,ID)	Obtain a lock to a record ID.
UNLOCKREC(cursor,ID)	Release a record ID lock.
READREC(cursor,ID)	Read a record from a file.
WRITEREC(cursor,ID,record)	Add or overwrite a record to a file.
DELETEREC(cursor,ID)	Remove a record from a file.
CALLAREXX(cursor,ID[,parms])	Call record as an ARexx macro.

Field functions

EXTRACTFIELD(rec,fld,itm,sub)	Get field or item from record.
STOREFIELD(rec,fld,itm,sub,str)	Put field or item into record.
INSERTFIELD(rec,fld,itm,sub,str)	Insert field or item into record.
DELETEFIELD(rec,fld,itm,sub)	Remove a field or item from record.
LOCATEFIELD(str,rec,delim)	Search fields for a string.
LOCATEFIELDBY(str,rec,delim,srt)	Search sorted fields.
REDUCEFIELDS(record,fields)	Extract multiple fields.
TRANSPOSEFIELDS(record)	Matrix transpose fields/items.
SORTFIELDS(record,delim,srt)	Sort fields in a record.

String functions

CAPITALIZE(string)	Convert all but first char to LC.
CONVERT(string,from,to)	Convert characters in a string.
COUNTCHAR(string,char)	Count a particular character.
QUOTE(string)	Encapsulate string in quotes.
TRIMSP(string)	Strip blanks from a string.

Stem functions

SEPARATE(name,delim[,pat])	Convert record into stem.
COMBINE(name)	Convert stem into record.
READSTEM(crsr,ID,stem[,delim[,pat]])	Read record to stem.
WRITESTEM(cursor,ID,stem)	Write stem as record.

Indexing functions

CREATEINDEX(cursor,name,keysrt,idsrt[,nodesize])	Create an index.
OPENINDEX(cursor[,name])	Get cursor for support part or index.
DELETEINDEX(icursor)	Delete an index.
CLEARINDEX(icursor)	Re-initialize an index.
WRITEKEY(icursor,key,ID)	Add a key/ID to an index.
DELETEKEY(icursor,key,ID)	Delete a key/ID from an index.
GETFIRSTID(cursor[,key])	Get first ID in index or file.
GETNEXTID(cursor)	Get next ID in index or file.
GETPREVID(icursor)	Get previous ID in index.
GETLASTID(icursor[,key])	Get last ID in index.
GETKEY(icursor)	Get key for previous ID.

"WRITEREC", "DELETEREC", and "CLEARFILE" are already in the VLTPhones file. They are also included separately on the disk for reference by those who don't have RexxDB.) Start up RexxDB and VLT. In VLT, assign "VLTPhones.vlt" to any key. Now, when you press that key, the VLTPhones window will come up.

VLTPhones is just a starting point. With the functionality provided by RexxDB, VLT, and ARexx, this script can be expanded to meet all of your telecommunication needs. RexxDB could store a script with each phone number to run when the phone is answered, that would log on to the BBS for you. You can add scripts to keep a database of other BBS'ers, and store old messages. You can even use multitasking to have one script down-loading messages from one BBS, while you read the messages previously downloaded from another. Does any other computer make this possible?

ARexxDB and RexxDB are trademarks of JMH Software of Minnesota Inc.

For more information contact JMH Software of Minnesota Inc., 7200 Hemlock Lane, Maple Grove, MN 55369, (612)424-5464.

ARexx, $49.95, William S. Hawes, P.O. Box 308, Maynard, MA 01754. ☑

Part One: Understanding Ray Tracing

The Development of a Ray Tracer

by Bruno Costa

Amiga users have an uncommon familiarity with ray tracing. A little after the introduction of the machine, back in 1986, many Amiga users (and would-be users, like me) were amazed by the Juggler Demo, one of the first ray-traced animations in HAM mode, made by Dr. Eric Graham. The animation consisted of a man-like character, built just from spheres, juggling mirrored balls over a checkered ground—a typical ray-traced scene. It introduced three techniques that were mostly unused up to then: hold-and-modify mode, real-time animation playback from memory, and ray tracing.

Nowadays, the situation has changed a lot. There are at least a dozen packages that, among other things, produce and play real-time, ray-traced animations in HAM mode. No other computer platform has such a wide usage of graphics rendering and animation software, and this might explain (or be explained by) the fact that the average Amiga user knows a good deal about graphics.

Most of us know that ray tracing is a technique for generating realistic computer images, with transparent and reflective surfaces and shadows, and that ray tracers are particularly obsessed with spheres. In this two-part article, I will try to explain this obsession, and also the theory of ray tracing, emphasizing on what it does well, what it doesn't, and why. I will also describe the practical aspects of implementing a ray tracer, demonstrating the theory and the programming suggestions with a full-featured implementation of an open-ended ray tracing package, exclusively distributed to AC's TECH readers. It is not in the public domain. This package, predictably called "ray," will allow you to generate your own ray-traced pictures, learn the ray tracing concepts, and apply the additional knowledge you might acquire in the recommended bibliography, adding or improving features of the ray tracer. If you are very practical and just wish to use the package to render some pictures, you might consider skipping directly to the Program Usage section.

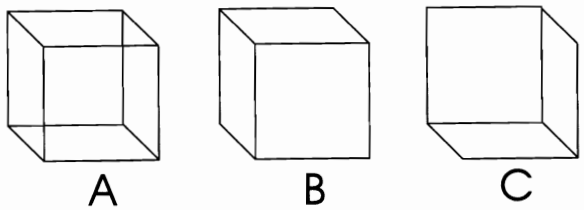
Ray tracing is a vast subject and, as such, it cannot be completely discussed even in a book-size manuscript. This article will try to introduce, or at least mention, many of the essential ideas and algorithms of ray tracing. Whenever appropriate, references to several book-size manuscripts will be made in the form of numbers inside square brackets. This is done in the hope that maybe a combination of all these books can provide a good notion of ray tracing. All the numbers in square brackets will be associated with the respective books in the Further Reading section, with entries containing enough information on each publication, including my personal comments on the contents and nature of the information in each text.

Introduction

The basic ray tracing algorithm can be seen as a very simple, elegant and ingenious solution to the so called hidden-surface removal, or visible-surface determination problem. This problem is one of the most fundamental in computer graphics, and consists simply of, given some kind of surface, how to determine which parts of it are visible. Its importance to realistic computer graphics can be better understood by looking at Figure 1a: you cannot tell for sure if the cube is positioned as in Fig. 1b or Fig. 1c. Most of the solutions to this problem are based on polygonal approximations of the surfaces; that is, instead of a perfectly round sphere, they use a group of polygons arranged in a sphere-like pattern. This practice does simplify the algorithms, but produces some undesirable effects that sometimes make the use of a rough approximation rather evident. Many of these visible-surface determination techniques are still in use, mainly for efficiency

→

Figure 1



reasons: ray tracing produces very good pictures in a long time, but some of the approximative techniques, when implemented in hardware, can generate "good enough" pictures in real-time.

To understand the ray tracing idea, let us assume that the screen is a plane, positioned in the three-dimensional world as in Figure 2, as though it was a photographic film inside a camera. The observer is somewhere just behind that camera, and the film will receive light from the scene. The ray-tracing solution to the visible-surface problem (Figure 2) is simply:

```
(each pixel in the screen)
{
        construct a line passing through the
        observer and this pixel;
        for (each object in the scene)
        calculate all interceptions of the line with the object;
        get the nearest of all the interceptions calculated above;
        render the current pixel;
}
```

The nearest interception is a point on the object that is visible when the observer looks toward that direction. The program may then render the pixel in question with the color of the visible object, using any one of several illumination models. Notice that ray tracing solves the visible-surface problem in a pixel-by-pixel basis, and this is one of the reasons why it is inherently slow. Note also that this very simple ray tracer already computes perspective effects: it does not use all those rotation and translation matrices in homogeneous coordinates that other algorithms require!

The algorithm above can be considered as a definition of ray tracing and, in fact, ray tracing as a solution of the visible-surface problem is just that, nothing more. Of course there are many optimizations of that simple algorithm that produce exactly the same pictures in much less time. There are also the illumination models, many kinds of anti-aliasing, shadows, reflections, refractions, textures, and so on. These additional topics are very important, so much that the variety and quality of these aids is what will determine the overall quality of a ray tracer.

Why Ray Tracing?

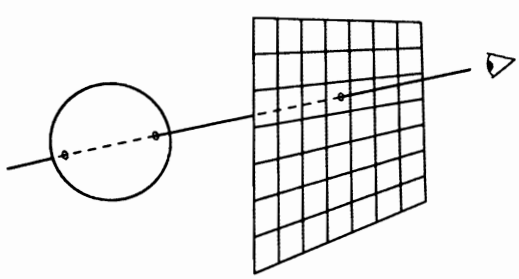
Certainly, the fact that ray tracing does not need polygonal approximations, although it can render them, is not enough to make it much better than the other alternatives. Why is it better? The answer lies in ray tracing's ability to deal with global illumination.

Essentially there are two kinds of illumination: global and local. The local illumination is simply the shading characteristics that depend just on local factors, such as attributes of the surface being shaded and of the light sources. Global illumination takes also into account the global factors, specially the influence of other objects in the scene. Rendering a mirrored or transparent surface needs global illumination, while shading of a wood or plastic object uses just local illumination.

The main reason why ray tracing is so good for global illumination calculations is its close relationship with the physics of light. In a sense, the ray tracing algorithm is a simulation of the macroscopic behavior of light, based on real-world optics.

The concept of ray tracing came from the idea of following the path of each light ray in a scene to determine which objects are illuminated and how, which rays bounce in which objects, and finally which subset of the many rays that leave a lamp reaches the observer. Obviously, such a calculation would be infinitely expensive, since most of the rays that leave light sources never reach the observer. However, there is a physical principle, obvious at first, that allows ray tracing to exist and be computationally viable: the reversibility of the light rays. It says simply that the path followed by a light ray is the same regardless of the way you trace it; e.g., if in Figure 5 a ray coming from point P and passing through point O will pass through point Q, then a ray from Q passing through O will certainly pass through P, following the same path as the former ray. This simple principle allows us to trace the rays in a backwards way. We start from the observer instead of the lamps, calculating just the rays that reach the screen. Thus, all the rays we trace are relevant for the picture being rendered.

Figure 2



We can try to understand why this reversibility works for ray tracing using a more intuitive explanation. To see an object, your eyes must catch some light coming from that object. Most objects do not produce light themselves and so, to see them, there must be some kind of light source nearby, which will emit rays that will be reflected by the objects around. The whole idea of ray tracing starts at the observer to determine the color of each pixel in the screen. We know that whatever color the observer sees a particular pixel, it was stimulated by a light ray coming in a straight line passing through that pixel. So we trace this possible ray, to see if there was any surface in that direction that could have sent some light. If this ray hits a surface, then there was some light coming from that direction, and so we compute, through the local illumination model, which color that point on the surface was. During this step, we check the position of this point on the surface relative to the light sources, to determine the intensity and color of the light that reaches this surface, if any, and is sent to the observer. If we are considering global effects, such as reflections and refractions, we must check, by tracing additional rays, if any light came from the reflection or refraction directions. If so, the results of these rays are combined with the local color in that surface point to determine the color of the screen pixel in question. The above steps are repeated for each pixel on the screen. You should notice that the rays still come from the light sources. They are just traced in a backwards fashion, which is much more efficient—in fact it is feasible—and produces the same results.

Color Algebra

To fully understand illumination models one must understand what is color, how it is represented, and how it is combined with other colors. There are many answers for the first two questions, and an excellent assortment of them can be found in some of the references[4,5].

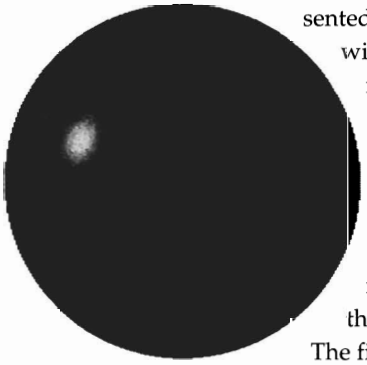


Figure 3

There are two simple and interesting facts about color that should be mentioned here. The first is for the mathematically inclined: the color space is a vector space. This means that we can manipulate colors as if they were represented by vectors: we can add and subtract them using vector rules, compute linear combinations, measure the distance between two colors, change the vector space basis and so on. The second is that color, as perceived by the human eye, can be represented as a three-dimensional space; that is to say, the combinations of just three "linearly independent" colors produce all the colors humans can perceive. Combining two colors is not enough, and at the same time with four colors one is redundant. The RGB system uses red, green, and blue as basis of the vector space, but other systems may use different basis; e.g., the YMC system uses yellow, magenta and, cyan.

Most, if not all, of the commercial rendering packages use the RGB representation of color. It uses three numbers to represent a color: an amount of red, an amount of green, and an amount of blue that, when added together, produce the desired color. Obviously, the red, green, and blue used to define the other colors must be precisely defined and standardized. The RGB representation contains information on both the chromaticity of the color and its brightness: (1,0,0) is red, (2,0,0) is also red, but brighter, and (0,0,1) is blue. It is important to remember that the RGB representation is quite far from the ideal, and that it sometimes may produce some unreal effects. It has the advantages of being very simple and specially compatible with the current display technology. Besides, for most mundane rendering tasks, it is quite acceptable.
To our needs, it is enough to know that we can multiply an RGB color by a constant, add or subtract two colors, and even calculate dot and cross products, just as if colors were represented by a 3-D vector.

A Simple Local Illumination Model

The basic ray tracing algorithm can determine which object is visible in each pixel of the screen. To compute the actual color of the pixels, we must have some way to simulate or approximate the behavior of light when it reaches a surface. The rules and equations that describe this simulation are called the illumination model. I won't delve into much detail about surface physics; it is a vast and complex matter that is fully described in the references[4,2,3]. This section will describe essentially a simple illumination model developed by Whitted, but some of the concepts presented below will be useful in uderstanding other advanced models.

There are mainly three illumination factors that contribute to the local illumination in the Whitted model: the diffuse contribution, the specular contribution, and the ambient contribution. It is possible to distinguish between the effects of these contributions in Figure 3. The diffuse contribution is the portion of light that is absorbed by the surface, scattered and then re-emitted in random directions. It produces the uniform

→

shading, the smooth change from the light to the dark side of each sphere. The ambient illumination also produces diffuse effects, but it is like a giant light source that contains the whole scene. It is an approximation of the effect of the infinitely many light sources or reflectors that surround us in the real world, for instance, the light entering through a window or reflected by hundreds of unimportant objects that do not appear in a scene. It contributes to the diffuse illumination of the whole surface of the objects, but is specially important in the parts that are not illuminated by any light source, since in these places ambient illumination is the only effective contribution. The specular contribution is the light that is reflected directly by the outermost boundary of the surface, in contrast with the diffuse illumination that is first superficially absorbed. It produces the highlights on the spheres in the picture. Notice that the specular reflection does have a predominant direction; i.e., it is much stronger when the observer is near the direction of reflection of rays from a light source. It depends on the position of the observer. You should check that by moving around objects that have highlights. You will notice that the points that are highlighted in the surface will move. In fact, if you have some billiard balls around, you might take the time for an unforgettable surface physics practical lesson. Just don't let anyone see you playing with the balls!

To be used in computer graphics, the contributions described above must be simulated or reproduced using some kind of mathematical model, usually in the form of one or more equations. This model is normally an approximation of the real thing, either based on the physical theory behind it, or just on plain empirical observations. In fact, computer graphics_and other physical sciences—advancements are often based on this kind of research: from observations of nature, we determine what our models are missing. Then we try to establish some rules that describe better that behavior of the real world. Finally, we incorporate, if practical, this new description, or an approximation of it, to the existing work.

The Whitted model is mostly an approximation of the real world based on practical observations: it just tries to make things look right. It can be summarized by the five equations in Figure 4.

Figure 4

(1) $C_A = I_a$

(2) $C_D = \sum_{j=1}^{nlamps} \frac{I_{ij} \cos\theta_j}{d_j + K}$

(3) $C_S = \sum_{j=1}^{nlamps} \frac{I_{ij} \cos^n \alpha_j}{d_j + K}$

(4) $C_L = k_a C_A + k_d C_D + k_s C_S$

(5) $I = k_l C_L + k_r C_R + k_t C_T$

Contributions:

C_A = ambient contribution
C_D = diffuse contribution
C_S = specular contribution
C_L = local contribution
C_R = reflection contribution
C_T = transmission contribution

Object dependent constants:

k_a = ambient contribution factor
k_d = diffuse contribution factor
k_s = specular contribution factor
n = specular concentration factor
k_l = local contribution factor
k_r = reflection contribution factor
k_t = transmission contribution factor

Lamp dependent constants:

I_{ij} = intensity of the lamp
d_j = distance from the lamp to the object
θ_j = angle between lamp direction and surface normal

Lamp and Observer dependent constants:

α_j = angle between the observer direction and the reflected ray

Global constants:

I_a = intensity of the ambient illumination
K = arbitrary light attenuation constant (usually 1.0)

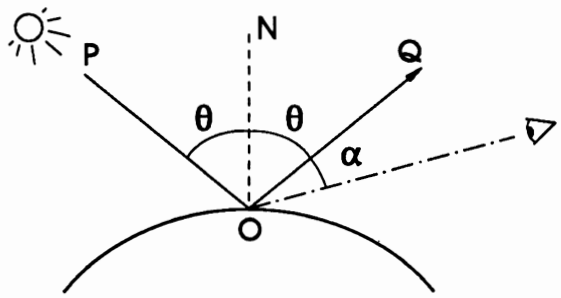
Equation 1 describes the ambient illumination. It is simply an RGB vector that measures the overall ambient illumination color in the scene.

Equation 2 is for the diffuse contribution of all the lamps. As you might have observed in your surface physics practical lesson—remember, it was unforgettable!— the parts of an object that receive light directly are brighter than parts that receive light in an oblique angle. This behavior is modelled in Equation 2 with the cosine of the angle between the lamp direction and the surface normal (Figure 5): when the angle is zero (light reaching the surface perpendicularly) the cosine is 1 (full illumination); with a 90 degree angle, the cosine is 0 (no contribution, since the light passes directly); other angles in-between produce cosines ranging from 0 to 1. This is multiplied by the color of the lamp in question, to work as an

attenuation factor that will control the influence of this lamp on the surface. There is an additional effect that is considered both in the diffuse and specular contributions: the decay of light energy with increasing distances. Far away lamps are surely not as important as nearer ones. It is known from physics that the luminance is inversely proportional to the square of the distance—for point light sources. We should then divide the diffuse and specular contributions by the square of the distance from the lamp to the surface. Although it has been tried, the results were not satisfactory, mainly for two reasons: if the distance was smaller than 1.0, the intensity of the light would increase tremendously, and the square of the distance produced a very fast decay that made pictures look too dark. Since this is an empirical model, Whitted simply dropped the square, and added a constant to the distance to assure that it would be always greater than one. This modified rule resulted in some of the first nice ray-traced pictures.

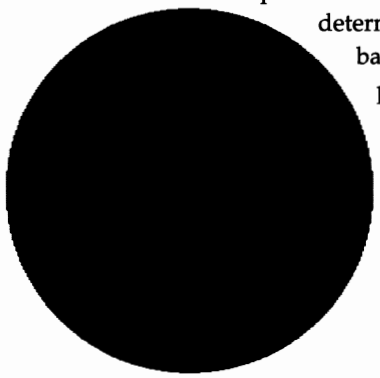
The specular contribution is described by Equation 3. As mentioned before, the specular contribution of each lamp is divided by the distance to the light source to model the energy decay. It is also multiplied by the color of the light source and an additional attenuation factor that will control the influence of the lamp. In this case, since we want to model highlights, we want the influence to be high in some restricted area, and

Figure 5

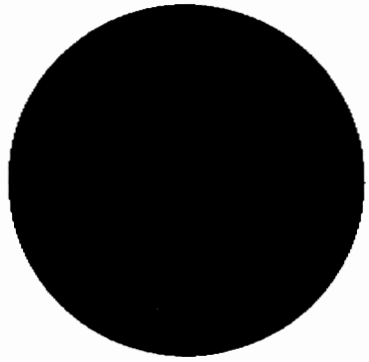


almost null in the rest. This behavior is obtained by the cosine of the angle between the direction of the observer and the direction of reflection of the light coming from that lamp (Figure 5). When this angle is small, the observer is near the direction of reflection, and thus receives a good amount of specularly reflected light from the surface. (The cosine is almost one.) As the observer moves away from this reflection direction, the angle will increase, the cosine will decrease and the influence will decline considerably. The cosine alone produces too wide highlights, and an exponent has been added to concentrate them: now, the attenuation factor will be near one just for directions very close to the reflection direction, and virtually zero everywhere else. This concentration may be controlled by varying the value of n, as you can see comparing the results in Figure 6.

Equation 4 combines the effects of the three contributions above to produce the local illumination contribution, using a simple weighted average. The sum of the weights should be one, in such a way that this would be a true weighted average, but experiments have shown that, sometimes, values slightly off this limit would produce better effects. These weights are determined in an object-by-object basis, and careful choices can produce realistic effects. The average in Equation 5 will be used to model the global illumination, described just below.



n = 10



n = 40

Figure 6

Global Illumination Effects

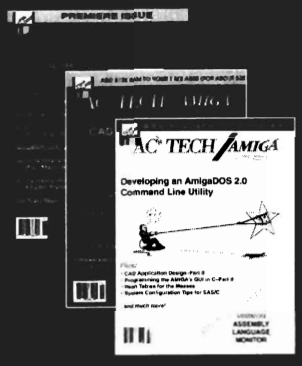
To obtain reflections and refractions of light rays, we must use the corresponding laws of optics. Let us assume that the surface in Figure 7 is partially reflective and partially transparent, and that the line MN is perpendicular to it—that is, normal to the surface). When the incident ray (I) reaches the surface, part of it is reflected (R) in such a way that the angle of reflection is the same as the angle of incidence. Part of the remaining energy of the ray is refracted (T) and the equation in the picture relates the angle of refraction to both the angle of incidence and the indices of refraction of each media. The remaining energy is either lost—converted to heat, etc.—or produces the local illumination effects.

The simple ray tracing algorithm can be extended to compute global illumination factors in a logical way:

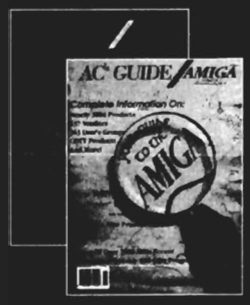
• When a ray hits a reflective or refractive surface, we do not stop there: it is refracted and/or reflected and followed further.

Amazing
Covers the AMIGA

Amazing Computing
For The Commodore AMIGA



AC's TECH
For The Commodore AMIGA



AC's GUIDE
To The Commodore AMIGA

AC's publications have always been innovative and complete. With the Premiere issue of ***Amazing Computing*** in February 1986, we introduced the first monthly magazine dedicated to the Amiga. AC's commitment to deliver solid information and valuable insight for the Amiga continues today. AC remains the first in news coverage—often providing complete stories and pictures of fast-breaking Amiga events in the next issue. AC is a forerunner in providing a well balanced mix of reviews, tutorials, tips, programming tasks, hardware projects, and more. Each issue of *Amazing Computing For The Commodore Amiga* is packed with the best of the Amiga.

AC's TECH For The Commodore AMIGA is the first and largest publication dedicated to the technical promise of the Amiga. Each quarterly issue provides new frontiers for the Amiga user eager to do more. *AC's TECH* not only attempts to define what the Amiga can do, but expands those boundaries.

AC's GUIDE To The Commodore AMIGA is the first and only complete guide to the Commodore Amiga. *AC's GUIDE* is the one resource used by the entire Amiga industry for Amiga product information. Yet *AC's GUIDE* also offers a listing of freely redistributable software and a growing registry of Amiga user's groups. *AC's GUIDE* is the complete resource to the expanding platform of Amiga products and services.

If you are not an AC subscriber, you don't know what you're missing. AC's publications are produced to give you more choices and resources. AC makes sure that whatever is happening in the Amiga market, you'll know about it.

To order a subscription, please use the order forms in this issue or for credit card orders, call toll-free

1-800-345-3360

from anywhere in the U.S. & Canada.

Don't Just Sit There, *SUBSCRIBE!*

AC's TECH/AMIGA

For the Commodore

AC's TECH For The Commodore Amiga is the first disk-based technical magazine for the Amiga, and it remains the best. Each issue explores the Amiga in an in-depth manner unavailable anywhere else. From hardware articles to programming techniques, *AC's TECH* is a fundamental resource for every Amiga user who wants to understand the Amiga and improve its performance.

AC's TECH offers its readers an expanding reference of Amiga technical knowledge. As the Amiga enters an assortment of new markets and new possibilities, *AC's TECH* is there to provide support to Amiga users and programmers. If you are constantly challenged by the possibilities of the world's most adaptable computer, read the publication that delivers the best in technical insight, *AC's TECH For The Commodore Amiga*.

Save!

Charter Subscription Offer

4 Big Issues — Just $39.95

(limited time only)

Use the subscription card here
or use your Master Card
or Visa and call

1-800-345-3360.

Hurry!
Charter Sub Rates Deadline
November 30, 1991

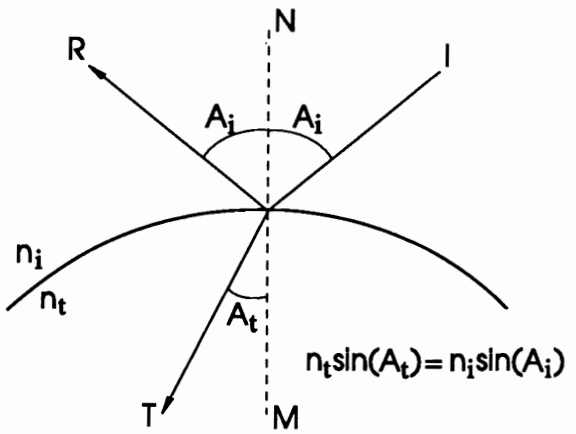
• If the surface is reflective, we construct a new ray belonging to the same plane as the normal and the incident ray, forming an angle with the normal equal to the angle of incidence. We apply the ray tracing procedure recursively to this ray to determine the contribution of reflection to the color of this point in the surface.

• If the surface is transparent, we construct a new ray belonging to the same plane as the normal and the incident ray, forming an angle with the normal calculated by the formula in Figure 7. We apply the ray-tracing procedure recursively to this ray to determine the contribution of refraction to the color of this point in the surface. Note that it is very important to keep track of the changes in the refraction coefficients when there are changes of transmitting medium.

• The refraction and reflection contributions are combined with the local illumination factors to determine the color of this point in the surface. This combination is proportional to the amount of energy that was transferred in each way; e.g., if most of the energy was reflected, then the reflection contribution is the most important. In the Whitted model, this combination is done in a very simple way, as described by Equation 5. It is another weighted average, where the weights—the proportion by which the light energy is split—are constant for each object and given by some of their attributes. Please notice that this approximation, although acceptable in most pictures with carefully fine-tuned parameters, is certainly distant from the reality. If you look at a transparent plastic or glass flat surface at angles varying from zero to almost 90 degrees, you will notice that it will slowly change from a perfectly transparent plate to an almost perfect mirror. The real proportion between reflected and refracted light is a function of the angle of incidence and the color of the incoming light, and it is not easy to calculate. It is given by the Fresnel relationship, fully explained in some of the references[4,1].

You might have noticed that the steps above form a recursive procedure and, as such, there must exist a way to stop the recursion. We can use a recursion count variable that is incremented each time the recursive routine is entered and decremented every time it is exited. If this variable exceeds a certain limit, the routine returns immediately, and the recursion will come to an end. (This is exactly what is done in the implementation on disk.) To reach the recursion limit usually—but not always—means that further calculations will not contribute too much to the original pixel ray being calculated. There are other ways of stopping the recursion, based on an estimated "importance" of the ray: when it drops under a certain lower limit, the recursion is stopped. This latter procedure, although seemingly clever, has some problems that make it less attractive. It is described in the references[1,3].

Figure 7



Shadows

The addition of shadows to ray tracing with global illumination is absolutely straightforward. The shadow determination is a form of visibility problem: points that are not visible from a lamp are in the shadow. In other words, a point in a surface is in the shadow of a lamp if that lamp is not visible from that point. Thus, when rendering a point in a surface, we simply shoot a ray in the direction of each lamp and trace it: if it hits any object, that lamp will not be considered in the computation of the illumination of that point. That's all there is to it.

The Importance of Interceptions

From the ray tracing algorithm it is possible to conclude that everything one needs to know about an object to render it, is how to calculate the interception of a given line with that object. Well, this is essentially true, except for some additional attributes that are needed for shading and creating other effects. Particularly important is the ability to calculate the normal to the surface at each point, since it is indispensable for even the most basic shading. The fact that you just have to know how to solve the interception equation to fully determine the shape of an object is one of the major strengths of ray tracing. This equation is usually simple, even for some mathematical objects that are so complex that they cannot be

visualized properly using other techniques. Also, the interception calculation is specially simple for spheres, and that, you guessed, is the reason why so many ray tracers like to render spheres—they are simpler and faster, and look very nice reflecting things!

Program Usage

This section explains, from the user point of view, how the ray tracer provided on disk works. It is an implementation of many of the ideas presented here, and it is the main subject of the continuation of this article in the next issue, where the full source code for it will be given.

Included on the disk you will find four executables: "ray", "ray24", "ray.881" and "ray24.881". Those with the .881 extension need at least a 68020/68881 processor pair, and the others work on any Amiga. Versions with and without 68881 support are absolutely identical, but be careful not to run the .881 version in a non-accelerated Amiga, since a visit of the guru will be unavoidable. The executables with the 24 suffix produce 24-bit true color output, and those without will render to an interlaced HAM screen and save standard IFF files.

"Ray" runs from the CLI only, supporting a good set of command line options. If you type "ray -h" the following help message will be printed to the CLI:

```
usage:
ray [-dstrah?][-Rxyz<n>][-p<x>/<y>][-o<name>]<scene file>
-d = disables dithering
-s = disables shadows
-t = disables transparency
-r = disables reflection
-a = disables anti-aliasing
-R = sets max recursion level to 'n'
-x = sets picture x-size
-y = sets picture y-size
-z = sets picture depth
-p = forces x-y aspect ratio to 'x'/'y'
-o = output rendering data to file 'name'
-h = this help\dS<normal>
```

The only required argument is a scene file name. "Ray" works based on a very simple scene file that contains the description of the scene to be rendered, including all the objects and its attributes, the observer and the lamps. This scene file and the CLI-only interface are not really part of the ray tracer; they are just the front-end by which the program is presented to the user. In this case, a portable, primitive but powerful front-end was desired, but nothing prevents one to write a fully intuition-based interface, that allows the user to manipulate objects and set their attributes using the mouse. Alternatively, a shell, a new front-end over an existing program or library, that produced scene files could be written. It would allow the user to build scenes interactively, and then save them or, better yet, run "ray" directly and transparently from inside it. It will be seen in the next part of this article that the sole existence of this scene file conflicts with the object-oriented design of the program: there are conceptually better solutions that are, unfortunately, somewhat less convenient. The format of the scene file is detailed on the disk, including many useful examples and images.

Most of the options of "ray" are self-evident, but some are worth a couple of comments. First of all, the executables on disk do not support anti-aliasing, transparency, and picture depth. The respective options are for future expansion. Dithering is a technique, fully described in Part 2, used to mix available colors to produce an apparent increase in the number of them. The option to set the maximum recursion level affects multiple reflections, like when there are two mirrors facing each other. This value will limit the number of inter-reflections, that would be infinite otherwise.

The HAM version of the program automatically detects the aspect ratio of the screen pixels, and thus doesn't need the -p option, although a different aspect ratio can be forced if desired. This version of the program renders directly to a HAM custom screen that pops up when the program starts. When the rendering is over, you may click the right mouse button to exit immediately or the left to save a HAM IFF picture to the file given by the -o option. Rendering in any version of ray may be interrupted safely by hitting Ctrl-C at the CLI.

The aspect ratio control option is specially important for the 24-bit driver, since it does not know the aspect ratio of the pixels of the target device. Depending on your usage of the produced files, you may use different aspect ratios. If they are to be displayed in HAM, 88/100; for interlaced HAM, 177/100; for publication, 1/1 may be better, since you can scale pictures in your DTP program. This version outputs a RAY1 24-bit file directly to the filename specified by the -o option. This file must be converted to IFF24 using the supplied utility raytoiff. Support for different 24-bit file formats is as easy as providing another separate utility like this. (The source code will be provided in the next issue.) Note that the output of raytoiff is compatible with The Art Department, but it doesn't seem to work well with Professional Page 2.0. I do not have the IFF24 specification so I couldn't determine exactly why. As an ankward workaround, you might consider loading the files in TAD and resaving them to produce fully compatible pictures. Better yet, you might fix raytoiff, or even write another converter, to produce fully compliant 24-bit IFF files—but don't forget to send it to me!

In the next part of this article, many of the practical aspects of ray tracing will be discussed, showing an application of the theory presented here. The source code of a complete ray tracer will be commented and explained, design decisions justified, and programming techniques revealed. Particularly remarkable issues are the portability of the program, the object-oriented philosophy, and the extensibility of the implementation. These three characteristics are specially important, since they make it easy to add new kinds of objects to the ray tracer, and also allow the program to be used in other machines with different operating systems. Hopefully, this implementation will help you to understand the theory and find out by yourself many of the difficulties and satisfactions of implementing a ray tracer. See you next issue!

Further Reading

[1]. Foley, A. van Dam, S. Feiner & J. Hughes, *Computer Graphics, Principles and Practice* , Second Edition; Addison-Wesley, 1990.

It could be called the current computer-graphics bible. It is a huge hard-cover book, with up-to-date information on everything about graphics, including many algorithms, graphics hardware technology and user interface design. It has been significantly enhanced over the first edition, but, as you might suspect, it does not have everything, although it has references to all the topics that are not fully explained. It is an expensive book that you should buy if you are really interested in computer graphics.

[2] A. Glassner, J. Arvo, R. Cook, E. Haines, P. Hanrahan, P. Heckbert & D. Kirk, *An Introduction to Ray Tracing*; Academic Press, 1989.

By far the best and most complete reference on ray tracing, this book is based on an introductory course given in the SIGGRAPH conferences of 1987 and 1988 by the authors, but it was significantly revised and expanded. It contains good, tutorial information on basic ray tracing algorithms, and a unique ray tracing implementation guide. I just wish we had seen it before starting to write our ray tracer—things would be much more easier.

[3] D. Rodgers, *Procedural Elements for Computer Graphics*; McGraw-Hill, 1985.

It is a good, general reference for computer graphics algorithms. In spite of starting to look a little outdated by now, it still helps to situate ray tracing among other rendering techniques.

[4] R. Hall, *Illumination and Color in Computer -Generated Imagery*; Springer-Verlag, 1988.

Currently the best reference dedicated to the complex subject of illumination and color. Mr. Hall is one of the most prominent researchers in this area, and, although his book contains information not easily found elsewhere, I usually have a hard time trying to locate and understand the information he provides!

[5] A. Glassner, J. Arvo, R. Hall, J. Kajiya, D. Mitchell & J. Wallace, *Advanced Topics in Ray Tracing* ; SIGGRAPH'90 course notes, August 1990.

This is an advanced text, in fact a collection of papers by approximately the same authors of *An Introduction to Ray Tracing*. In this course, it was recommended that attendants had a good mathematical background and previous experience with ray tracing, preferably having implemented at least one ray tracer. Particularly, I think that after reading the two parts of this article and understanding the provided ray tracer, it is quite possible to comprehend at least a portion of it. I don't know if this publication is very easy to obtain, but if you want to improve this ray tracer, it has some excellent tips and ideas. It also contains an extensive ray tracing bibliography.

[6]R. Jaeschke, *Portability and the C Language* ; Hayden Books, 1988.

A book every C programmer should have. It defines precisely the ANSI C standard, and also gives all the details of previous dialects and implementations of C, particularly Kernighan & Ritchie C in different UNIX systems, VAX/VMS, and MS-DOS operating systems. If you develop some code that is not closely tied to the operating system under it, you should consider writing it in a portable fashion to allow it to run in many more machines. Be careful not to believe too much in everything Mr. Jaeschke says. Perfect portability is utopian. People should expect just code with a high portability degree.

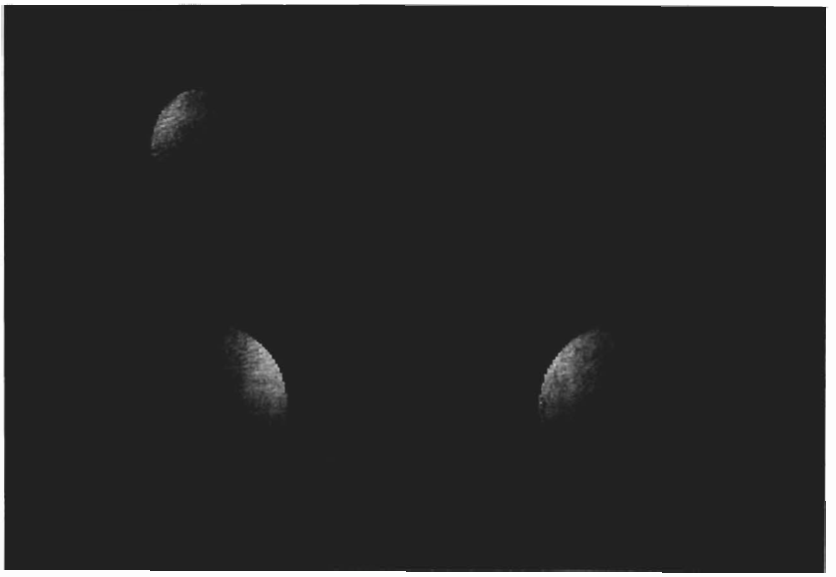
About The Author

Bruno Costa is a computer engineering student at PUC/RJ and works with systems programming and computer graphics at the IMPA (Institute of Pure and Applied Mathematics) in Rio de Janeiro, Brazil. He has owned Amiga computers since 1987. Contact him on Internet as bruno@brlncc.bitnet.

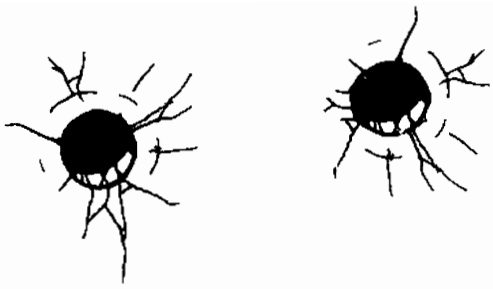
Please write to Bruno Costa,

c/o AC's TECH
Post Office Box 869
Fall River, MA 02720-0869

We'll forward the information to Bruno in Brazil, ASAP!



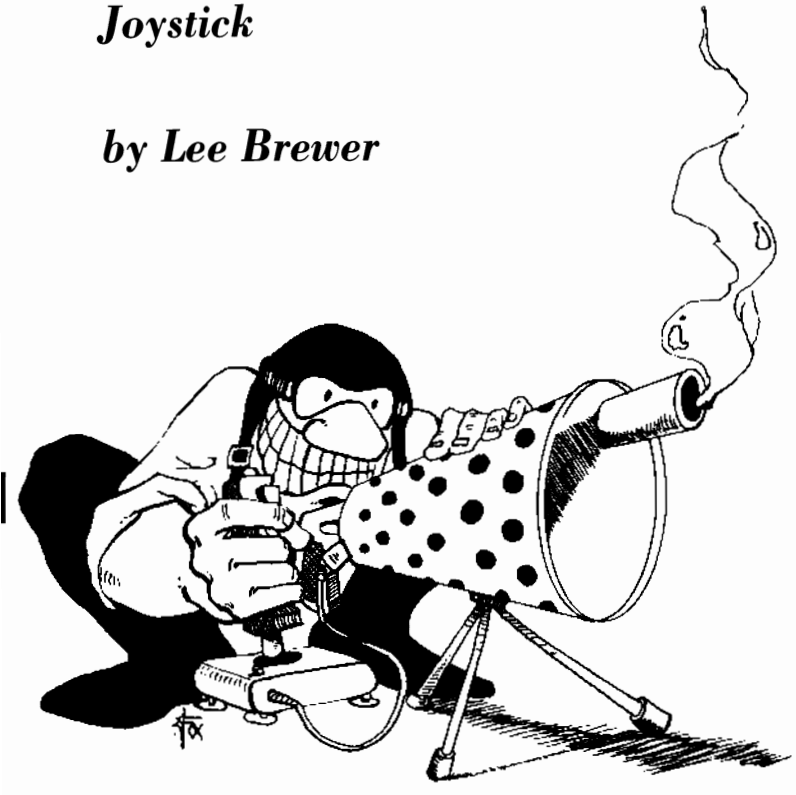
COLORLAMPS.ILBM by Bruno Costa



The Varafire Solution

Add Variable-Rapid Fire Capabilities to Your Favorite Joystick

by Lee Brewer



Disclaimer

Although I am sure there is no cause for worry, I guess I had better include a disclaimer here that the author is in no way responsible for problems which may arise from the use of the modified joystick. This disclaimer includes re-addiction to some of those old games you have tucked away in your box of long-forgotten disks!

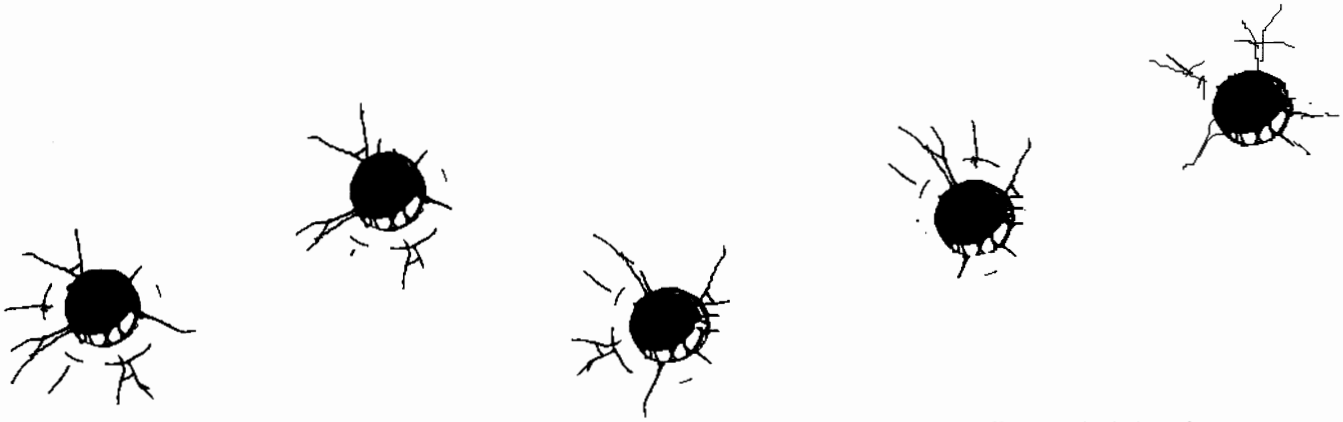
The Problem

You know you are close. Yes! Here comes the last wave of those who would stop you from reaching your goal. Fearlessly, you face your enemies, for you have met them many times. For others who have never been here before, it is said that this wave of attackers is impossible. . . but not to you. Your experience, hardened skills, and dexterity quickly rescue you from what would appear—to others—to be certain doom.

Then silence comes. You know this silence well. You realize it marks the time when your ultimate foe is gearing up to release his siege of awesome fury. As you feel the sweat gathering on your forehead, the question once again races in your mind, "Will this at last be the time that victory will smile upon me?" You brace yourself, tighten your grip, and *it* appears! You start your offensive maneuvers and shoot as fast as you can! The menace cringes with pain as your shots strike home! You are firing so fast that your hand starts aching, but you realize you don't dare quit or the normal zillion rays of deadly energy beams flare and—all at once—"Game Over, Game Over!" Does the crazy programmer of this game think I have a bionic thumb or something? No one can shoot *that* fast!

Yes, you and I have both been there. Whether he thought winning would kill the challenge and make the game uninteresting, or he just plain had a mean streak when he coded it, the programmer of the game never intended for *anyone* to see the ending screen. Cheer up, video warriors! There is an answer!

Learn basic circuit design and project building techniques with this simple, yet in-depth, beginner's tutorial.



The Solution

This article is written to offer a practical, low-cost solution to anyone who wishes the fire button couldbe used a little, or much, faster when playing this or her favorite game. Don't quit reading now just because you find my approach different from other rapid fire devices.

In the past I have seen, purchased, and been disappointed by various rapid fire devices. Some of these mechanisms did not fire as fast as I could without using them and three others I purchased didn't work at all!

Being annoyed, I decided to construct a working rapid fire device of my own. However, there were times I found different games allowed my ship/player only a certain maximum number of shots on the screen at one time. Term this maximum number the "shot-allowance" of the game. My standard rapid-fire device worked, but I still was killed off because the shots in my shot-allowance would come out as a solid beam. The rapid-fire device worked too well Although I had a longer energy bolt to fire at the enemy, some of them could more readily dodge my fire, since it was now only one shot rather than four or five, and I still got killed.

This circumstance helped me to come up with the idea of a variable rapid fire device. This mechanism allows me to synchronize my firing to the timing with which the enemies appear on the screen. It now is no problem that different games have different timing built into their waves of attackers. If my rapid fire device is too slow or too fast, I simply rotate a dial to select the desired timing of my shots. Now, none of the little beasties can get by, and finally the "impossible levels" are mine! In fact, once you build it, I suggest you try it on all the old games you conquered long ago. Your newly acquired super power will give the game a new level of excitement. (A side note here: Can you top the 25 times straight that I nailed Darth Vader's tie fighter in the "Star Wars" game by Broderbund?)

So What's Involved?

This project is one which requires you to do a modification on your joystick. When you are done you have a high-tech looking controller with both a dial and switch added to its case. You may also choose to add a longer cord to the joystick.

This project will take a little bit of time and precision work, but don't let either of these scare you off. I'm sure that after you have used the completed product, you will agree it was worth the effort. As for precision, if I can do it, anybody can.

There are two major parts to this operation. First is the replacement of the joystick cable. Second is the building of the circuit.

If you read through the last paragraph without a question popping into your head, then you probably are used to computer hardware modifications—or you were daydreaming. Most people are probably wondering what's wrong with the joystick cable the way it is? The answer. . .

Theory Behind Construction (Part 1)

To understand the reasoning behind the replacement of the cable, we must understand not only the output of the pins from the Amiga's game port, but also the way in which most joysticks are wired.

The Amiga has nine pins coming out of its game port, each of which has a number assigned to it and each of which performs a specific task. The task assigned to each numbered pin is as follows:

Amiga Game Port Pin Out Chart

Pin Number	Task
1	Controls Forward Movement
2	Controls Backward Movement
3	Controls Left Movement
4	Controls Right Movement
5	For Horizontal Movement w/ Paddle Controller
6	Controls Fire Button
7	+5 Volts (100 mA)
8	Ground
9	For Vertical Movement w/ Paddle Controller

Now that we have a pin-out chart for reference, let's see how this information relates to the joystick.

→

The joystick is simply a set of switches. There is a wire which runs from the computer's game port pin number 8 (ground), through the joystick cable, and then connects to every one of the joystick's internal switches. Out of the other side of each individual switch is a wire which runs back through the joystick cable to that switch's specific numbered pin of the computer's game port. When you make a move with the stick (for example, a right movement) the switch (number 4) closes the circuit between the computer pin (number 4) and the ground pin (number 8), thereby completing the circuit and producing the desired result of the computer being able to interpret your physical action.

Now we come to the root of why the old cable is inadequate. Most manufacturers do not make their cables such that there is a connection between the computers pin 7 and the joystick. Looking at the pin-out chart, we see that pin number 7 is a 5-volt power supply. Since there is no wire from pin number 7, there is no power supply going into the joystick. The main reason most joysticks lack this connection is probably that standard joysticks have no need for an internal power supply.

The absence of this connection poses a problem because our circuit, which will be internally installed in the joystick, needs 5 volts in order to operate. It is convenient that the Amiga provides a 5 volt supply coming off pin 7, but it is unfortunate that most cables does not allow pin 7 to connect to the joystick.

There may be a fortunate few, however, who have a cable which does connect to pin 7. If you look at the end of your joystick connecter and see a metal lead in hole number 7, then you might not have to worry about replacement. If you find a wire on the joystick side of the cable which you can discern is attached to pin 7, then you definitely don't need to replace your cable. However, if your stick is like most of them, the lead is not present, and you will need to add a new cable.

Theory Behind Construction Part 2

The second major part of the project is building the circuit. It is a relatively simple circuit having only one chip and a two other components. If you have never built a circuit before, give it a try. It's fun , and you may even find it addicting! However, if the common disease of microchip-itis gets the best of you, and you don't wish to attempt assembling the circuit yourself, see the end of the article where I will explain how you may obtain a pre-assembled circuit.

I also must note here that I used a WICO brand "Command Control" type joystick, typically known as a bat-handle, for this project. In this article I refer to this type of joystick. If you don't have one of these joysticks, don'tt despair. I'm sure that with a little creativity, you will be able to figure a way to adapt your joystick to the situation.

Also note that this modification may void the warranty, if any, on the joystick. However, I have used my Varafire for about two years and never had any problems with my machinery, computer, circuitry, etc.

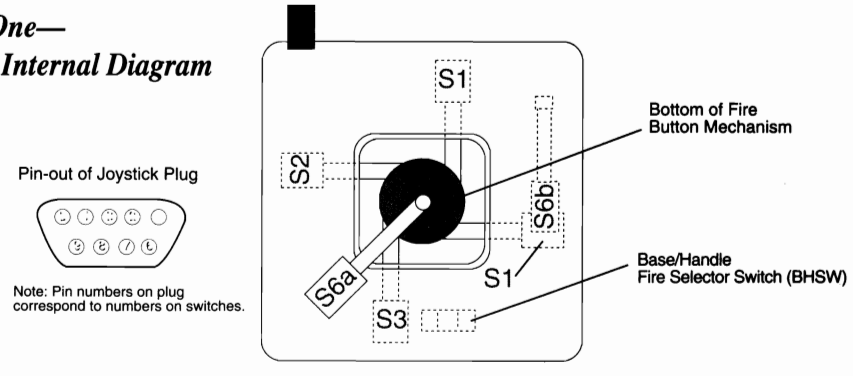
Circuit Board Theory of Operation:

There are a few of you who probably will want to know how the circuit actually operates. If curiosity gets you, read on. The connection made from pin 7 of the Amiga provides 5 volts DC for the 555 timer chip. The user-installed switch toggles between single shot input from the user or pulsed input from the 555 timer. When the chip is engaged, output from the chip's pin 3 (which is a pulsating current), is sent to the switch. So when the user holds the switch closed the circuit simulates rapid pulses of shots just as if the user were opening and closing the switch very rapidly. And of course what makes the Varafire unique from similar devices is the potentiometer. This component attaches between the chip's pin number 7 and number 8. The tasks of these pins, respectively, are discharge and ground. Upon rotation the potentiometer varies the amount of voltage input to the chip and thereby increases or decreases the rate of the pulses the circuit outputs.

The Cable Procedure

Disassemble your joystick by removing the screws which hold its top to its bottom. Lay the bottom aside and look at the inside of the top piece. Rotate the joystick until you are faced with a scene similar to Figure 1.

Figure One—
Joystick Internal Diagram



Explanation of Figure 1:

In Figure 1, we are faced with a picture of the inside of the joystick mechanism. This mechanism is attached to the underside of the joystick housing's top. The rectangles labeled S2, S4, etc. are the switches which sense movement. Please note the position of S1 as being underneath S6b. Also note the rectangle placed to the

right of S3 as being the switch (labeled Base/Handle Fire Selector Switch or BHSW), which allows the user to toggle between use of the fire button, which is installed in the base or the fire button installed in the handle. The three vertical lines in BHSW represent the three contacts protruding from the bottom of this switch. The shaded circle in the center is, as meant by its label, the plastic disk which is the bottom part of the handle-mounted fire button mechanism.

Step 1: Labeling

Place small pieces of masking tape on the bottom of each of the switches. With a marker, label each piece of tape corresponding to the labels on the bottoms of the switches in figure 1. If you were paying attention, you probably will have noted that the numbers on the labels correspond to the number of the game port pin which that switch connects with.

Now an important side note: for simplification I will refer to the individual switches by their labels and also will refer to the wires by the number appearing on the switch which they attach to, or by the number of the game port pin to which they are attached. I will term the switches as S1, S2, etc. I will term the wires as W1, W2, etc.

Returning to Figure 1, you will see that S6a and S6b are both fire buttons. Since this joystick allows either base or handle options for firing, there obviously must be two switches attached to pin number 6.

Step2: Cable Connections

Strip enough insulation off one end of your 9-wire cable to expose the concealed wires. Strip off approximately 1/4 inch of insulation from each of the wires. Solder these wires onto the back of the DB9 connector in any order. As you solder, make a DB9 pin-to-wire chart on your paper (you did see 'paper' in the parts list didn't you?) by listing each DB9 pin number and the color of the corresponding wire you solder to it. This is very important as you will need this chart for reference later.

Disconnect the old cable from the joystick housing. Cut each of the old wires in such a way that you have enough wire left over to splice your new cable wires onto them. Inside your joystick you will note that S6b has a wire connecting it to S6a (I will call it wire 6a-b), and S6a has another wire going from it to the joystick cable. Don't cut wire 6a-b. Cut only the wire from S6a to the cable and then use this for the splice.

Strip three to four inches of insulation off the unused end of your new cable. Take each of the old wires in turn and individually trace each one from its contact on its switch until you reach its free end. Use your DB9 pin-to-wire chart as a reference to help you splice the appropriate numbered new wire with its old counterpart. For example, the wire attached to S1 should have its free end spliced to the new wire which is attached to pin number 1 of the DB9.

Parts List

This is what you will need to build your Varafire rapid fire joystick controller

For the joystick:

1. A length of nine-conductor cable. The length depends on how long you wish your joystick cable to be. I recommend 6.5 feet.
   Radio Shack Cat. #278-775. $.59/ft
2. A nine-position female D-subminiature connector (DB9).
   Radio Shack Cat. #276-1428. $1.19
3. A nine-position female D-subminiature connector Housing. This catalog number is for a plastic one, but metal ones, such as I used on mine, are available.
   Radio Shack Cat. #276-1539. $.79
4. Four 6-inch pieces of small diameter wire, such as telephone wire, to connect the completed circuit board to the potentiometer and switch which you install.

For the circuit:

1. One 555 timer integrated circuit chip (not the CMOS version).
   Radio Shack Cat. #276-1723. $1.19
2. One 1uf 20v (or any voltage up to 30v) electrolytic capacitor I went to a small Radio Shack and was told they could order the capacitor, but most stores probably have this part on the shelf.
3. One 22 K Ohm resistor.
   Radio Shack Cat. #271-1339 $.39
4. One 1 M Ohm potentiometer.
   Radio Shack Cat. #272-211 $1.19
5. One 5A 125V SPDT toggle switch.
   Radio Shack Cat. #275-603 $1.59
   or #275-635 $2.89

Other:

7. Masking tape.
8. Magic marker.
9. Pen and paper.
10. Electrical Tape.
11. Solder and soldering iron.

→

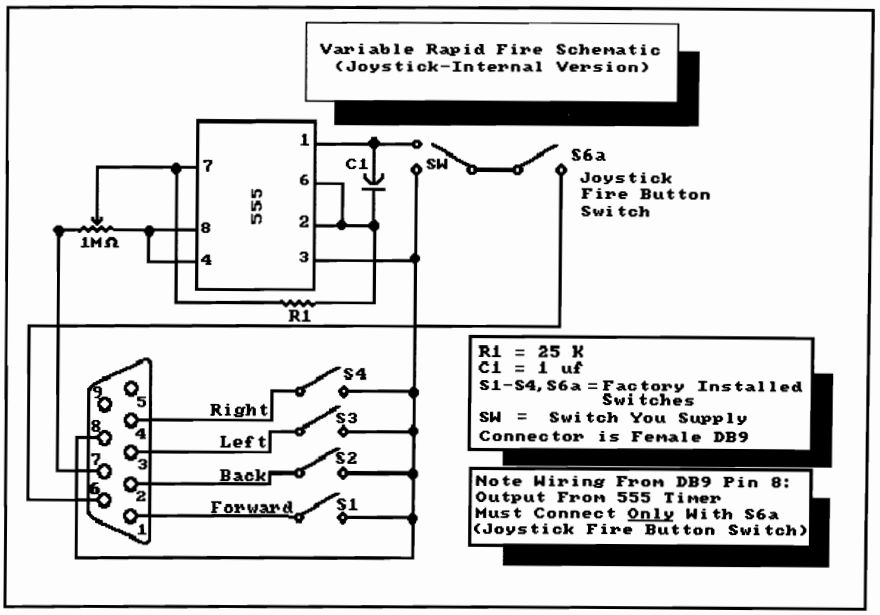


Figure 2

Step 1: Circuit Building

Build your circuit board according to the enclosed schematic (see Figure 2). The only additional note I can add to aid in the understanding of Figure 2 is for inexperienced circuit builders. Remember that the actual pin numbers on your 555 timer chip are numbered differently than what appears on the schematic. To find pin number 1 on the 555, hold the chip right side up in front of you, and rotate it until the indented circle is in the upper left. The top left-most pin is number 1. The rest of the pin numbers may be figured by counterclockwise counting the pins.

If you like to etch printed circuit boards, then refer to the included PCB pattern (see Figure 3a.) There is a process by which, utilizing a photocopy machine, you may transfer the PCB image onto a sheet of TEC-200 film. You then use a normal iron to transfer the pattern from the film onto a copper-clad circuit board. Use ferric chloride etchant, drill the proper holes, and you are ready to assemble the circuit.

Refer to Figure 3b for parts placement. Please note that this figure is drawn such that it displays how the components are to be placed when the PC board is oriented exactly how it appears in Figure 3a.

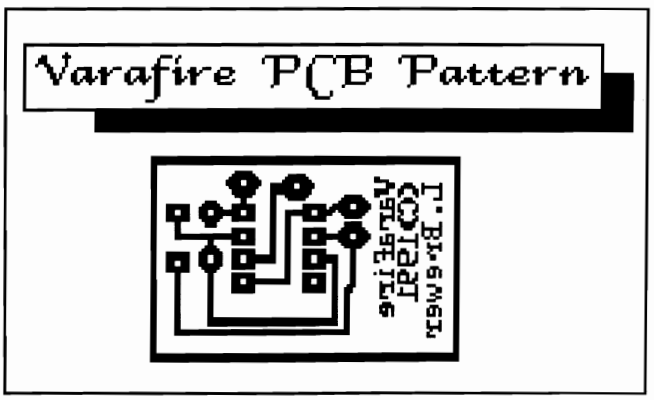


Figure 3A

Step 2: Installation of SW and Pt

Refer to Figure 4 and note that SW (the SPDT switch you will use to toggle your Varafire on/off) is the vertical rectangle in the lower left of the diagram that has three horizontal boxes drawn in it, and several arrows pointing to it. As before, these three boxes represent the switch's contacts. Also note that the potentiometer is in the upper left and is labeled "Pt." The locations shown are not mandatory, of course, but are suggestions which make good use of the limited space available inside the joystick housing.

You will see that the path of W8 has been traced to allow those who are curious to figure out the logic of the two fire buttons. Note also that BHSW has been included as in figure 1. For a complete understanding of this project, it is important to note how BHSW figures into the circuit. This switch allows the Varafire to work with both the base and handle fire buttons.

Drill appropriate-sized holes in the joystick housing for SW and the potentiometer. Insert these two components, and, utilizing Figure 4 (and 3b if needed), properly connect the two wires, PW1 and PW2, to the potentiometer. You will note that PW1 and PW2 do not connect to each other. PW2 must be soldered to Pt's middle pole and PW1 may be attached to either of the outer poles of Pt. The decision of which pole to

choose is based upon which direction you wish to rotate your potentiometer knob so as to increase/decrease your firing rate. In a Varafire where PW1 is connected to the pole closest to the joystick cable (as shown in Figure 4), a clockwise rotation of Pt decreases firing speed. If PW1 is attached to the pole farthest from the cable, clockwise rotation yields a firing speed increase. Whichever pole you choose, make sure the same pole also has W7 from the cable soldered to it.

When you solder your wires A and B onto SW, carefully check that you have properly traced the paths of these wires so the correct connections are made. Note that wires A and B do not connect with each other.

Step 3: More Wiring!

Next comes a step which will sound confusing although it is very simple. Study Figure 4 as you read this next paragraph and the procedure should become very clear.

Now we come to the splitting of W8. Two wires are attached to S6a. One of these wires is W6a-b and is not shown in Figure 4. Look inside your stick, locate W6a-b, and make sure you grab the other wire coming off S6a. The wire you are holding is actually an extension of W8 which attaches S6a to the left-most pole of BHSW. Trace this W8 extension 1.5"-2" back from its connection to S6a and cut the wire. Take the piece of the W8 extension which is still attached to S6a, trace it to its newly cut end and solder it to the middle pole of SW (see Figure 4). The other cut piece of the W8 extension must be soldered onto SW at the same place you attach wire A from the 555 timer circuit (again see Figure 4), thereby making a connection between SW and BHSW.

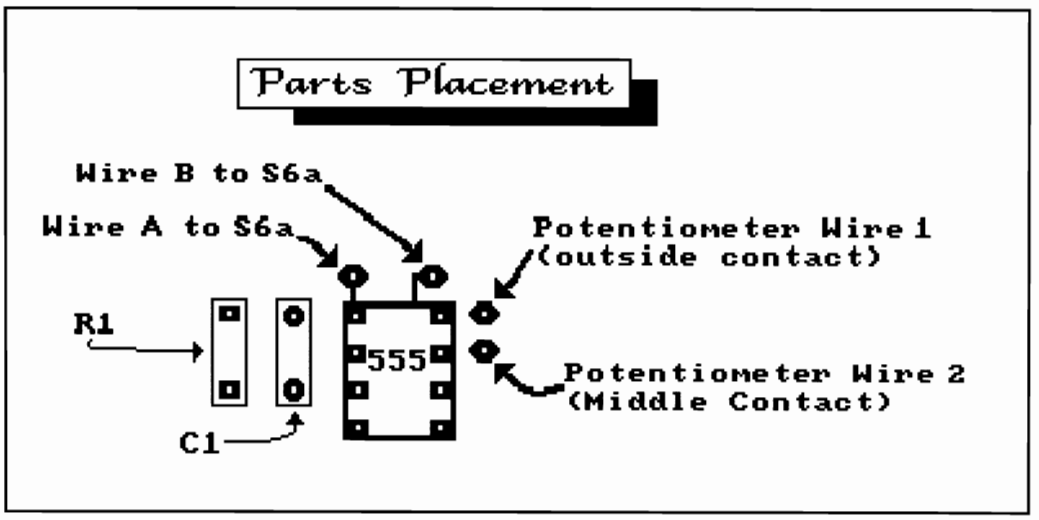


Figure 3B

Step 4: Installing the Circuit Board

The last step to be performed is the placing of the circuit board into the joystick. I used a piece of double-faced "foam tape" to attach the bottom of my circuitboard to the bottom of the potentiometer. I made sure that the foam tape covered the circuit's entire bottom surface so that no shorts could occur between any of its components and the metal bottom of the potentiometer.

Step 5: Re-assembly

Put the cable in the groove designed for it in the joystick housing. I suggest a small piece of the foam tape being wrapped around it to help it stay in place. Look into each of the screw's holes to make sure no wires have gotten themselves into the pathway the screws will take (Murphy's law, you know.) Also, check that no wires are sticking out of the sides (you wouldn't want to have to re-splice them would you ?) Insert and tighten the screws.

Wrap Up

I hope that using your Varafire will bring you many hours of fun. I also hope it will spark a new flame into those old games you thought you would never touch again.

If you are one who does not like to build circuits, then I can provide one. Send me $20.00 to cover parts, S&H etc., and I will build the circuit (composed of R1; C1; the 555 timer chip; wires A, B, PW1, PW2; Pt; and SW) for you.

Notes

Ten sheets of TEC-200 image film for making printed circuit board transfers with a laser printer or photocopier is for $5.95 plus $3 p/h from DC Electronics, Box 3203, Scottsdale, AZ 85271, (800)423-0070

The PCB pattern is laid out for an axial, not radial, type capacitor.

Special thanks go to Marc Kuntz, without whom this project would not have been possible.

About the Author

Lee Brewer is a math/computer science teacher. He started out building hardware projects for a Commodore 64 several years ago and now does the same for his Amiga. He can be reached c/o *AC's TECH*.

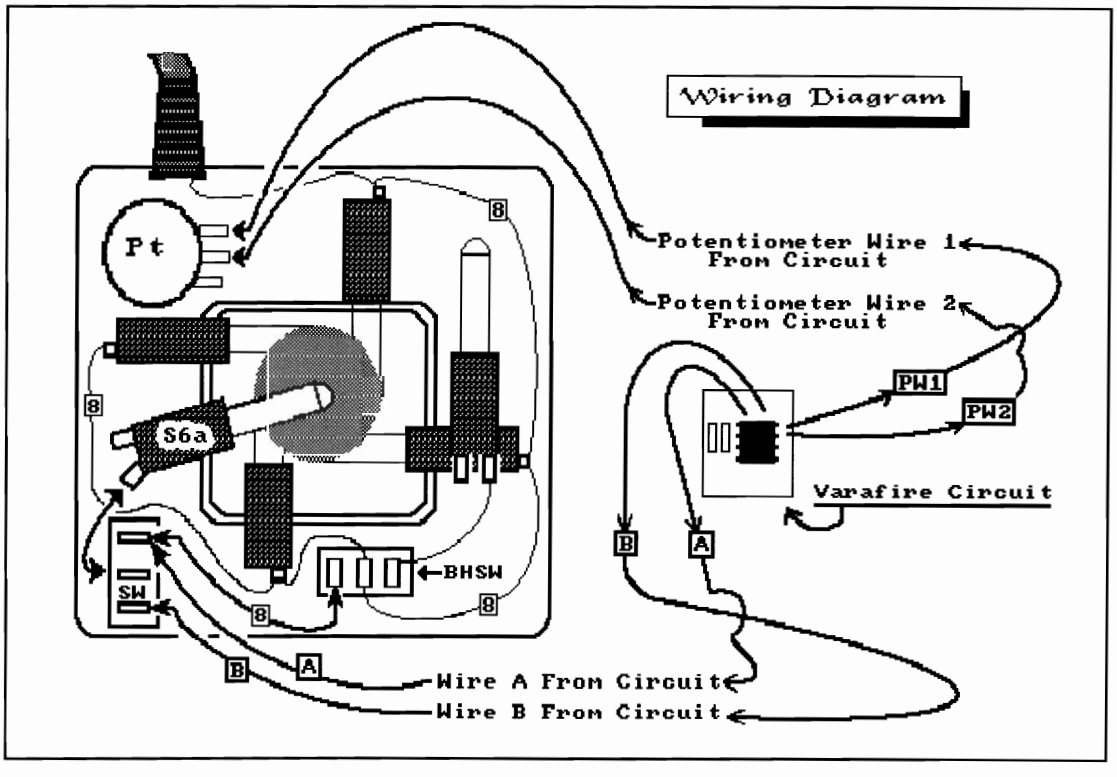


Figure 4

Part Three: Structuring A Display Module

Programming The Amiga's GUI*

*Graphical User Interface

This is the third in a series of articles that will help you take advantage of many of the custom features of your Amiga using the C programming language. In this issue you will find:

- *A discussion of the most important aspect of structured programming, information hiding.*
- *A presentation of a structured method for opening windows using a handle, a pointer to a pointer.*
- *A discussion of the opening of screens.*
- *Instructions for using a pre-compiled display module to open screens and windows.*
- *A recommended programming shell with hidden functions.*
- *An introduction to drawing graphic images, including a simple rosette.*

by Paul Castonguay

Last issue you saw a rather detailed discussion about how to best use your compiler to work on projects consisting of many pre- compiled modules. Hopefully you now understand the crucial role played by the LMK program and feel comfortable using it. It turns out that much of your success in C on the Amiga rests on your ability to use such tools. You can know all the theory in the world, but if you can't competitively use your development environment, you won't ever get much done. In this issue, I move the emphasis from the actual compiler to the C language itself. However, I continue to focus on best ways to access features that are particular to the Amiga.

PROGRAM STRUCTURE

Some hobbyists think that a structured language is one that offers the programmer various features like FOR-LOOPs and SWITCHes. Although these features (often called constructs) are desirable, they do not necessarily lead to good program structure. To demonstrate this I will first compare two different constructs in the same programming application. Then I will structure the program. You will see that the two ideas, constructs and program structure, are quite different.

A SORTING EXAMPLE

A program that sorts a series of numbers may take advantage of the automatic incrementing of the FOR-LOOP to conveniently accomplish its work, yet the same algorithm can be achieved using the more general WHILE-LOOP which offers no such convenience. In the second case the programmer simply declares and increments his own loop counter to accomplish the same thing. The following implementations of the popular bubble sort algorithm demonstrate my point:

An introduction to drawing graphic images including how to draw this rosette.

```
/* bubble sort using for-loop */

int bubble(int my_array[], int how_many)
{
   int inner, outer, temp;

   for(outer = how_many-1; outer > 0; --outer)
   {
      for(inner = how_many-1; inner > ((how_many-1) - outer);
          --inner)
      {
         if(my_array[inner] < my_array[inner-1])
         {
            temp = my_array[inner];
            my_array[inner] = my_array[inner-1];
            my_array[inner-1] = temp;
         }
      }
   }
}


/* bubble sort using while-loop */

int bubble(int my_array[], int how_many)
{
   int inner, outer, temp;

   outer = how_many-1;
   inner = how_many-1;
   while(outer > 0)
   {
      inner = how_many-1;
      while(inner > (how_many-1 - outer)
      {
         if(my_array[inner] < my_array[inner-1])
         {
            temp = my_array[inner];
            my_array[inner] = my_array[inner-1];
            my_array[inner-1] = temp;
         }
         --inner;
      }
      --outer;
   }
}
```

The first version, using the FOR-LOOP, is certainly preferable. It concentrates all the working parts of the loop into one place, the parentheses of the FOR-LOOP, reducing the chance of missing an integral part of its operation. In the WHILE-LOOP version, the loop counters are incremented at the end of each block, separated from where they are initialized. In a longer program they might be forgotten, creating an infinite loop. Thus, the FOR-LOOP construct offers advantages over the WHILE-LOOP equivalent, and it is common to say that the first version is better structured. But in this case the word structure is referring only to one block of code, not to the prøgram as a whole. This is NOT what is generally meant by the term "Structured Programming."

TOP DOWN DESIGN

The structure of a program refers to how it is divided up into parts, a process often referred to as "top down design", or "step-wise refinement." The idea is to divide a complex problem into lessor, easier to handle sub-problems which can be later joined together to accomplish a final, complete solution. This method is not unique to computer programming. It is used for problem-solving in many fields.

In almost any language a program can be either well structured or badly structured. It all depends on how you, the programmer, choose to divide it into parts. The trick is to realize exactly what good program structure is, and how to achieve it.

INFORMATION HIDING

Splitting up a program has the advantage of shielding the often complex details of one part from the others. This is a formal principle in computer science called "information hiding." The term is often an anomaly to beginning programming students. How can concealing parts of a problem improve its readability? Sounds silly, I know, but it is true. It's all a question of what you hide, and where. You want to hide those parts that are not needed by others, but also in such a way that you can easily find them if and when you want to. Let me demonstrate by structuring the above bubble sort programs. That's right, neither above version is yet structured!

There are three instructions in the IF-BLOCK of the above example that together accomplish the interchanging of the contents of two elements of the array. Why three instructions? Because you must temporarily store one of the values in a safe place, otherwise you will loose it by overwriting during the swapping process.

```
a = b;      /* Value originally in a is lost.  */
b = a;      /* Variables are equal. */


temp = a;   /* Value originally in a is saved. */
a = b;
b = temp;   /* Variables are swapped. */
```

But suppose you did not know this. Maybe you think that computers should be smart enough to understand such things on their own. You might have always relied on the SWAP function of AmigaBASIC and never had to think any deeper about exactly how a computer achieves a swap between two variables.

The following modified solution structures the program by dividing it into two major tasks, that of sorting the numbers and that of swapping any two numbers.

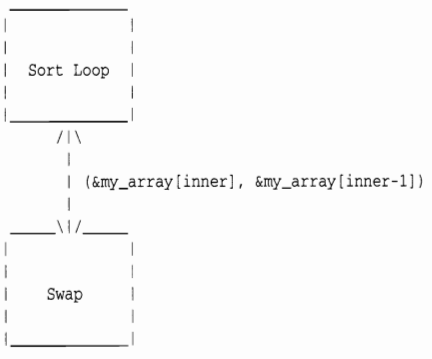
```
/* Structured Bubble sort using for-loop */

int sort(int my_array[], int how_many)
{
   int inner, outer, temp;

   for(outer = how_many-1; outer > 0; --outer)
   {
      for(inner = how_many-1; inner > ((how_many-1) - outer);
          --inner)
         if(my_array[inner] < my_array[inner-1])
            swap(&my_array[inner], &my_array[inner-1];
   }
}
```

```
VOID swap(int *first, int *second)
{
   int temp;

   temp = *first;
   *first = *second;
   *second = *temp;
}
```

The task of swapping is subservient to that of sorting in that it is called by it. The intricate details of swapping are completely hidden from the part of the program that does the sorting, which is now free to concern itself only with sorting. The main() function does not know for instance that to swap two numbers a temporary variable is used. Similarly, the swap routine doesn't know the exact purpose of the program. It concerns itself only with the swapping of numbers. On the other hand, some information is shared between them. The sorting part passes the addresses of the numbers it wants exchanged and the swapping part changes them directly using pointers.

The result of all this is that the main program is simplified. Also, the program as a whole is easier to maintain. You might, for instance, want to add a feature to report whenever a swap occurs. Simply design a new function to accomplish that and add a single call to it in the IF-BLOCK of the sorting loop.

The graphical representation of a structured program requires a new kind of diagram. Below I present a hierarchical diagram that shows the relationship between the two parts of our example.



Here the sorting part is shown calling the swapping part, perhaps repeatedly. The arrows indicates that information is handed back and forth between the two through its argument list. A flow chart is still needed, of course, to show the internal workings of each block, but often a complex program consists of so many blocks, each containing so many other underlying blocks sharing and hiding information differently, that a hierarchical diagram is crucial to gaining any comprehension of the operation as a whole.

To summarize, structured programming has very little to do, if anything at all, with FOR-LOOPs and SWITCHes. Rather it has to do with functions and subroutines, and how they are used to either hide or share various details upon which the operation of a program depends. It turns out that much of the published code for the Amiga does not follow the above important programming guidelines. That is why I am presenting it here in detail. In fact, many supposedly "technical" publications completely misrepresent the idea of structured programming by limiting themselves only to a discussion of constructs. I saw one recently that described the act of structuring a program as an equivalent to removing its program labels (tie points for goto statements). That's ridiculous. In contrast, I am taking the trouble to provide you with the most accurate information possible, and backing it up with accepted principles in computer science.

STRUCTURING A WINDOW OPENING FUNCTION

Last issue you saw an unstructured example of a window opening program. Below I give you the same one in structured form, where all details except for the valuable window pointer are hidden. Naturally you would not design this function for a real application. All it does is open the same simple window every time. I use it here to familiarize you with what is involved in converting this operation from unstructured to structured form. At the same time, don't under-estimate its complexity. It does contain a rather advanced technique that I explain below.

```
/*                        First_Window_Function.c                        */
/*                                                                       */
/*  AC's TECH AMIGA                        Volume 1, Number 4   */

#include <intuition/intuition.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/dos.h>
#include <stdio.h>
#include <stdlib.h>

#include "Libs.h"

VOID main(int argc, char *argv[]);
BOOL Open_Window_Function(struct Window *(*MyWindow));


VOID main(int argc, char *argv[])
{
   struct Window *MyWindow = NULL;

   if(!Open_Libs())
   {
      printf("Trouble opening libraries.\n");
      Delay(100);
      exit(RETURN_WARN);
   }

   if(!Open_Window_Function(&MyWindow))
   {
      printf("Trouble opening MyWindow.\n");
      Delay(100);
      Close_Libs();
      exit(RETURN_WARN);
   }
```

→

Amazing AMIGA COMPUTING

AC's GUIDE AMIGA AC's TECH AMIGA

Name _____

Address _____

City _____ State _____ ZIP _____

Charge my ☐ Visa ☐ MC # _____

Expiration Date _____ Signature _____

Please circle to indicate this is a **New Subscription** or a **Renewal**

MasterCard ® VISA®

PROPER ADDRESS REQUIRED: In order to expedite and guarantee your order, all large Public Domain Software Orders, as well as most Back Issue orders, are shipped by United Parcel Service. UPS requires that all packages be addressed to a street address for correct delivery.
PAYMENTS BY CHECK: All payments made by check or money order must be in US funds drawn on a U.S. bank.

One Year of Amazing!	**Save over 49%** 12 monthly issues of the number one resource to the Commodore Amiga, ***Amazing Computing*** at a savings of over $23.00 off the newsstand price!	☐ ☐ ☐	US $24.00 Canada/Mexico $34.00 Foreign Surface $44.00
One Year of AC SuperSub!	**Save over 46%** 12 monthly issues of Amazing Computing PLUS AC's GUIDE AMIGA 2 Product Guides a year! A savings of over $31.30 off the newsstand price!	☐ ☐ ☐	US $36.00 Canada/Mexico $54.00 Foreign Surface $64.00
Two Years of Amazing!	**Save over 59%** 24 monthly issues of the number one resource to the Commodore Amiga, Amazing Computing at a savings of over $56.80 off the newsstand price!	☐	US $38.00 (sorry no foreign orders available at this frequency)
Two Years of AC SuperSub!	**Save over 56%** 24 monthly issues of Amazing Computing PLUS AC's GUIDE AMIGA 4 Product Guides! A savings of over $75.60 off the newsstand price!	☐	US $59.00 (sorry no foreign orders available at this frequency)

Please circle any additional choices below: *(Domestic and Foreign air mail rates available on request)*

Back Issues: $5.00 each US, $6.00 each Canada and Mexico, $7.00 each Foreign Surface.

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
2.11	2.12	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	4.1	4.2	4.3	4.4	4.5
4.6	4.7	4.8	4.9	4.10	4.11	4.12	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12
6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9										

Back Issue Volumes: Volume 1-$19.95* Volume 2-$29.95* Volumes 3,4, or 5-$29.95* each
*All volume orders must include postage and handling charges: $4.00 each set US, $7.50 each set Canada and Mexico, and $10.00 each set for foreign surface orders. Air mail rates available.

AC's TECH AMIGA Single issues just $14.95! V1.1 (PREMIERE), V1.2, V1.3

Order a One-Year Subscription to *AC's TECH* Now – Get 4 BIG Issues!

Charter Rate Offer: $39.95 (limited time offer – US only)!

Canada & Mexico: $43.95 Foreign Surface: $47.95

Call or write for Air Mail rates!

Freely Distributable Software – Subscriber Special (yes, even the new ones!)

1 to 9 disks	**$6.00 each**
10 to 49 disks	**$5.00 each**
50 to 99 disks	**$4.00 each**
100 or more disks	**$3.00 each**

$7.00 each for non subscribers (three disk minimum on all foreign orders)

Amazing on Disk:
AC#1 ...Source & Listings V3.8 & V3.9
AC#2 ...Source & Listings V4.3 & V4.4
AC#3 ...Source & Listings V4.5 & V4.6
AC#4 ...Source & Listings V4.7 & V4.8
AC#5 ...Source & Listings V4.9
AC#6 ...Source & Listings V4.10 & V4.11
AC#7 ...Source & Listings V4.12 & V5.1
AC#8 ...Source & Listings V5.2 & 5.3
AC#9 ...Source & Listings V5.4 & V5.5
AC#10 ...Source & Listings V5.6 & 5.7
AC#11 ...Source & Listings V5.8, 5.9 & 5.10
AC#12 ...Source & Listings V5.11, 5.12 & 6.1
AC#13 ...Source & Listings V6.2 & 6.3
AC#14 ...Source & Listings V6.4, & 6.5

InNOCKulation Disk: IN#1 ...Virus protection
AC#15 ...Source & Listings V6.6, 6.7, 6.8, & 6.9

Please list your Freely Redistributable Software selections below:

AC Disks _____
(numbers 1 through 14)

AMICUS _____
(numbers 1 through 26)

Fred Fish Disks _____
(numbers 1 through 530; FF395 is currently unavailable. Please remember Fred Fish Disks 57, 80, & 87 have been removed from the collection)
You may FAX your order to 1-508-675-6002-

Subscription: $_____

Back Issues: $_____

AC's TECH: $_____

PDS Disks: $_____

Total: $_____
(subject to *applicable* sales tax)

Please complete this form and mail with check, money order or credit card information to:

P.i.M. Publications, Inc.
P.O. Box 869
Fall River, MA 02722-0869

Please allow 4 to 6 weeks for delivery of subscriptions in US.

Complete Today, or telephone 1-800-345-3360 now!

AC's TECH Disk
Volume 1, Number 4

A few notes before you dive into the disk!

- A working knowledge of the AmigaDOS CLI is necessary to get the most from this disk. Most of the files on the AC's TECH disk are only accessible through the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we have compressed many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). Lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh to RAM:*, from the CLI, type *lharc x foo RAM:* For help with lharc, type *lharc ?* *Archives with icons attached can be decompressed from the WorkBench by double-clicking the icon, and then suppling a destination path, when prompted.*

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 869
Fall River, MA 02720-0869

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc, their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas)

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications, or any of it's agents, is not responsible for any damages incurred while attempting these projects.

```
    SetAPen(MyWindow->RPort, 1);
    Move(MyWindow->RPort, 93, 50);
    Text(MyWindow->RPort, "Help me, I'm stuck in here.\n", 27);

    Delay(500);

    if(MyWindow)
       CloseWindow(MyWindow);

    Close_Libs();

    exit(RETURN_OK);
}



BOOL Open_Window_Function(struct Window *(*wnd))
{
    struct NewWindow MyNewWindow;

    MyNewWindow.LeftEdge     = 100;
    MyNewWindow.TopEdge      = 50;
    MyNewWindow.Width        = 400;
    MyNewWindow.Height       = 100;
    MyNewWindow.DetailPen    = 3;
    MyNewWindow.BlockPen     = 2;
    MyNewWindow.Title        = "My First Window";
    MyNewWindow.Flags        = ACTIVATE | SMART_REFRESH;

    MyNewWindow.Screen       = NULL;
    MyNewWindow.Type         = WBENCHSCREEN;

    MyNewWindow.IDCMPFlags   = NULL;
    MyNewWindow.FirstGadget  = NULL;
    MyNewWindow.CheckMark    = NULL;
    MyNewWindow.BitMap       = NULL;
    MyNewWindow.MinWidth     = 0;
    MyNewWindow.MinHeight    = 0;
    MyNewWindow.MaxWidth     = 0;
    MyNewWindow.MaxHeight    = 0;

    if(!(*wnd=(struct Window *)OpenWindow(&MyNewWindow)))
       return(FALSE);
    else
       return(TRUE);
}
```

The above program is in the directory First_Window_Function on the enclosed diskette. It declares MyWindow as a pointer-to- struct-Window, initializes it to NULL, and passes its address to Open_Window_Function(). Presto, the address of your window gets deposited into the MyWindow pointer, like magic. Naturally you use the same popular calling style that we have been using all along, testing the function's returned value for success.

HANDLE - POINTER TO POINTER

Now get ready for some hand waving. From your work in Standard C, you should be comfortable with the practice of passing the address of a variable to a function when you want that variable to be modified by the function. That's what I did in our earlier swap function. See the excellent diagram and explanation of that same example in "The C Programming Language," second edition, by Brian Kernighan and Dennis Ritchie, Prentice Hall 1988 (referred to throughout this article series as K&R). In the main part of our above window example, we have the variable MyWindow, which represents a window. It makes no difference that it is a pointer variable, it is still a variable none the less and if we want Open_Window_Function() to change its value we had better pass its address, or forever hold its NULL.

On the function side, within Open_Window_Function(), things are a bit harder to see, but hang in there. The function receives an address as an argument, therefore it must be referenced by a pointer. Specifically it must be a pointer of the same data type as the corresponding variable in the calling function. Consider this slowly. The rule is, "Whatever the data type is in the calling function, you make a pointer to that type within the called function". In our example the data type of the variable in the calling function is pointer-to-struct-Window. I therefore must declare a pointer-to-pointer-to-struct-Window in the called function.

```
Declaration in Calling Function   Declaration in Called Function
 struct Window *MyWindow;              struct Window *(*wnd);
```

To change the contents of the calling function's variable from inside the called function, you must use the '*' operator on the variable you declared as its pointer. The following instruction (inside the called function) places the address of the window created by Intuition's OpenWindow() into the MyWindow pointer in the calling function by using the '*' character in front of the wnd variable:

```
*wnd = (struct Window *)OpenWindow(&MyNewWindow))
```

CONFUSED?

Consider that a pointer is a variable which, like any other variable, has both an address and a stored value.

```
        struct Window *MyWindow = NULL;

         ______________
        |              |
        |  MyWindow ------> ZERO ADDRESS
        |______________|


MyWindow  (contents of pointer) NULL
&MyWindow (address of pointer)  LEGAL ADDRESS determined by the compiler.
```

The address of a pointer is where it is located in memory and that is determined by the compiler. The stored value is a number that is placed at that address and that you have control over. Because our variable is a pointer, its stored value usually represents another address, or a NULL if it does not yet point to anything.

Now suppose I declare a second pointer, this time of type pointer-to-pointer-to-struct-Window.

```
struct Window *(*wnd) = NULL;

 ___________
|           |
|   wnd ------> ZERO ADDRESS
|___________|

wnd  (contents of pointer) NULL
&wnd (address of pointer)  LEGAL ADDRESS determined by the compiler.
```

This second pointer cannot point directly to a Window structure, it is not the correct type for that. It can only point to a pointer-to-struct-Window. So, I can assign to it the address of MyWindow:

```
wnd = &MyWindow;

 ___________      _____________
|           |    |             |
|  wnd ------>    |  MyWindow ------> ZERO ADDRESS
|___________|    |_____________|
```

That is essentially what happened in our above First_Window_Function() example when the argument &MyWindow was passed to the parameter wnd. The address of MyWindow got assigned to wnd. Wnd then pointed to MyWindow.

Just like with any pointer, I can use the '*' indirection or dereferencing operator on wnd to access whatever it is pointing to. Thus *wnd represents the contents of MyWindow (not the address of MyWindow, see K&R page 94), which happens to be NULL right now. If I assign something to *wnd, that something will go into MyWindow.

```
*wnd = OpenWindow(&NewWindow);

 ___________      _____________      ___________________
|           |    |             |    |                   |
|  wnd ------>    |  MyWindow ------>| Window structure  |
|___________|    |_____________|    | (newly created)   |
                                     |___________________|
```

Now the value stored in MyWindow has changed from NULL to an address determined by the Intuition's OpenWindow() function. It is now the address of the Window structure that was created. The value stored in wnd has not changed, it is still equal to the address of MyWindow. Don't get mixed up between the value stored in wnd and *wnd which is the value stored in what wnd is pointing to. Wnd still points to MyWindow! That is essentially what happened in our above First_Window_Function() example when the address returned by Intuition's OpenWindow() got assigned to *wnd, it changed the contents of MyWindow in main().

The term handle is used for a pointer-to-pointer because when you use it to call a function, you exchange not the object that you want to access, but something to which the object is or will be attached. It is like the handle of a frying pan that allows you to safely pick it up without touching the hot pan itself. In our programming example, we want to pick up a window, but we cannot do that directly at this time because it is not yet created. So we pass to the Open_Window_Function() a handle and ask that a window be attached to it.

WHY NOT AN ORDINARY POINTER

Perhaps you are thinking that it would be easier to declare the Open_Window_Function() function itself as type "pointer to Window" and have it return the newly created window's address directly. Although that might seem easier at first, it would be inconsistent, and consistency is another important aspect of structured programming. For the kinds of functions we are dealing with here, we want to design them to be of type integer, not some wierd pointer to structure, and to return a 1 for success, not an address. Besides, a function can return only one value. What are you going to do when you want your function to return two pointer values? Intuition is a pointer intensive environment where that requirement is bound to come up sooner or later, and when it does you will have to resort to the use of a handle anyway. Let's master the concept now and be prepared.

NOW FOR A REAL FUNCTION

Often the use of a computer's windowing system is tied in with whatever kind of application is being designed. An editor or word processor, for instance, would require its windows to have all the features the system can offer, borders, system gadgets, menus, etc. A CAD program might want a maximum of clear drawing area and not want some of these features. These articles do not revolve around the design of one single application; they are for hobbyists who are more interested in experimenting with programming in general.

Experience shows that hobbyists want to conveniently access all of the different graphic and color resolutions of the Amiga, but are less interested in window borders and system gadgets, at least at this early stage when they are trying to master the environment's more fundamental operations, like drawing graphics and displaying text. I have therefore designed a display module (Display.o) that contains functions that will allow you to do just that. To use it, simply pass your color and resolution requirements as arguments to one of its functions. In response you will always receive a full-size borderless drawing surface on which to render your graphics or text. This module, along with our earlier Libs.o, Libs.c, and Libs.h, are on disk in the directory called Support_Files. Parts of that code will be reproduced below as I explain its internal operation. However, before doing so I must say a few words about another graphic object on the Amiga, the Screen.

→

SCREENS AND WINDOWS

The Amiga provides different graphic/color resolutions through a graphic object called a Screen. Screens are the parent objects of windows, defining for them their graphic resolution and maximum number of colors. You should have a good understanding of this from your previous work in BASIC. AmigaBASIC parallels closely what I will be doing here.

The opening of a screen follows the same eight point operational plan presented last issue for opening windows. There is a NewScreen structure (a description structure) in which you enter all the characteristics of the Screen you want to create. Its template is in the header file <intuition/screens.h>. Here it is:

```
struct NewScreen
{
    SHORT LeftEdge;
    SHORT TopEdge;
    SHORT Width;
    SHORT Height;
    SHORT Depth;
    UBYTE DetailPen;
    UBYTE BlockPen;
    USHORT ViewModes;
    USHORT Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap;
};
```

The LeftEdge of a NewScreen structure must always be set to zero. The reason has to do with the fact that the Amiga cannot change display modes (resolution) within a single scan line. Thus each display line of a screen object must start at the left edge of the display and finish at the right. You probably already know this from the fact that you have never been able to drag a screen (using the mouse) horizontally, only vertically. Screens are always as wide as the display. TopEdge allows you to set the initial vertical position of a screen, allowing you to design displays having several different graphic resolutions. For instance, you could have a low resolution, 32 color graphic image in the top of the display and high resolution 80 column text in the lower part. In these articles I will always use zero for TopEdge, that is, I will always use the same graphic resolution for the entire display. Width must be set to either 320 or 640, corresponding to the Amiga's two horizontal graphic resolutions (low or high). Height must be set to either 200 or 400 (standard or interlace) for screens whose initial position is the top of the display.

For screens that start lower down (TopEdge greater than zero), you can set Height to less, but you must remember to make the sum of TopEdge and Height equal to or greater than the full height of the display. Depth is what sets the maximum number of colors and is inherited by all windows that are opened in a particular screen. You enter the number of bit-planes that produce the number of colors you want. You cannot ask for more than four bit-planes on a high-resolution screen, which gives you 16 colors, or 5 bit-planes on a low resolution screen, which gives you 32 colors. DetailPen and BlockPen mean the same thing as their equivalents in the NewWindow structure from last issue.

The ViewModes member must be set for the resolution you want your screen to have. The default is low resolution with a standard 200 lines. Use the macros HIRES for high-resolution and LACE for interlace. You can also specify SPRITES, DUALPF, HAM, EXTRA_HALFBRITE, and others, but I won't be covering these, unless you, the reader, ask that these articles be continued long enough to eventually cover those details. When you want to use more than one macro, separate them with C's bitwise OR operator, like this: HIRES | LACE. These macros are in the header file <graphics/view.h>. Set the Type member to CUSTOMSCREEN, a macro defined in the header file <intuition/screens.h>.

The Font member is a pointer to a structure that defines the default font of your screen and all windows that open in it. I will have more to say about that when I discuss how text is displayed in Intuition. For now simply set this member to NULL. DefaultTitle is a pointer to whatever text you want to appear in the title bar of your screen. I will not be using this since I will always be opening a full-size window that covers up the screen's title bar anyway. The Gadgets member is a pointer to a gadget structure of your own design. For now set this to NULL, meaning we have no custom gadgets for this screen. Actually the ROM KERNAL manual says that this member is not yet supported for screens. I have never tried to use it. Lastly we have CustomBitMap which is used in conjunction with the CUSTOMBITMAP macro of the above Type member. I will not be using this feature and will always set it to NULL. And that's it. Whew!

On the disk, you will find the program "First_Screen.c". It opens a single screen, leaves it open for about five seconds, then closes it. It is similar to the unstructured window program last issue. It also demonstrates how to display text directly in a screen (without a window), but I won't be doing much of that in these articles. Use the program to fiddle with some of the above members of the NewScreen structure.

OPENING SCREENS FROM DISPLAY.O

The operation of opening a screen has been structured as a function called Open_My_Screen(), and placed in the Display.o module. It uses three arguments: the resolution you want, the address of a pointer (a handle) to the screen you want, and the address of a pointer (a handle) to what is called the screen's ViewPort, which I talk about in a minute.

To specify resolutions, use the same vocabulary as is in True BASIC, "LACEHIGH16" for a high resolution, 16 color, interlace screen, "LOW32" for a low resolution, 32 color one, and so on. Refer to the instructions in the header file "Display.h" for how to call up all the other possible resolutions. You pass this argument to Open_My_Screen() as a string.

The second argument is the address of a previously declared pointer-to-struct-Screen. This is a handle, meaning that it will be modified by the function, receiving the address of the screen that it creates. This works exactly the same as the handle to a window explained earlier.

The last argument is the address of a previously declared pointer-to-struct-ViewPort, something that gets created by Intuition when it creates your custom screen. You will use this pointer in your programs to access your screen's color specifications. It is a handle, meaning that it will be modified by the function, receiving the address of the ViewPort that it creates.

You use the same popular calling style with Open_My_Screen() as you used earlier with Open_Libs(). The function returns, as you would expect, a 1 for success and 0 for failure.

```
struct Screen *my_scrn;
struct ViewPort *my_svp;

if(!(Open_My_Screen("LOW32", &my_scrn, &my_svp))
{
   ...

   take corrective action
   ...
}
```

Operation inside Open_My_Screen() is pretty straightforward. It starts by looking at the first argument (a string pointer) and tries to figure out what graphic resolution you want. All possible resolutions and color combinations are tested starting with HIGH2.

```
   if(strcmp(resolution, "HIGH2") == 0)
   {
      screen_description.Width     = 640;
      screen_description.Height    = 200;
      screen_description.Depth     = 1;
      screen_description.ViewModes = HIRES;
   }
   else if(strcmp(resolution, "HIGH4") == 0)
   {
      ...
      ...
```

The strcmp() function is described on page L219 of your SAS/C compiler manual. Here it determines if the string pointer, called resolution, points to the string "HIGH2". If so the related members of the NewScreen structure are assigned values accordingly. Otherwise, another possible resolution is tested. Eventually, if none of the resolutions are successful, the function returns with a FALSE value.

If a legitimate resolution is identified and assigned, the function continues and assigns all other members of the NewScreen structure. Notice the 0 for TopEdge, CUSTOMSCREEN for Type, and NULLs for complex things that we don't want.

```
   screen_description.LeftEdge     = 0;
   screen_description.TopEdge      = 0;
   screen_description.DetailPen    = 0;
   screen_description.BlockPen     = 1;
   screen_description.Type         = CUSTOMSCREEN;
   screen_description.Font         = NULL;
   screen_description.DefaultTitle = NULL;
   screen_description.Gadgets      = NULL;
   screen_description.CustomBitMap = NULL;
```

Finally the screen is opened by calling Intuition's OpenScreen() function, and the value returned is assigned to the my_screen handle.

```
if(!(*my_screen = (struct Screen *)OpenScreen(&screen_description)))
{
   printf("Cannot open screen!!\n");
   return(FALSE);
}
else
{
   *my_svp = &((*my_screen)->ViewPort);
   return(TRUE);
}
```

If the address returned by OpenScreen() is not legitimate the function returns a FALSE, otherwise it continues to perform one last operation, the assignment of the address of the screen's ViewPort to the my_svp handle. Following that, the function returns a TRUE.

OPENING WINDOWS FROM DISPLAY.O

Although it is possible to draw graphics and text directly in a screen, it is normally done through windows. For future examples in this series I provide the function Open_Blank_Window(). It takes four arguments, not all of which I will explain in this article. The first is a pointer to the screen in which you want the window to appear. This is not a handle. It is the address that you received when you used the Open_My_Screen() function. The Open_Blank_Window() function cannot modify that value. If you want to open a window on the Workbench screen you use a NULL for this first argument.

The second argument is a long unsigned integer (ULONG) which tells the function how you want to communicate with this window. We will leave this NULL today, meaning that we do not want to communicate with it at all. The third argument is the address of a previously declared pointer-to-struct-Window, and the fourth the address of a previously declared pointer-to-struct-RastPort, which I explain later. Both of these last arguments are handles, that is, the function will modify their values directly.

Here is the calling style, which you should be getting pretty used to by now:

```
struct Window *my_window = NULL;
struct RastPort *my_rp = NULL;
ULONG communication = NULL;

if(!Open_Blank_Window(my_screen, communication, &my_window, &my_rp))
{
   printf("Trouble opening window\n");
   ...

   take corrective action
   ...
}
```

→

The inside operation of this function is very similar to the screen opening function above. It fills in appropriate members of a NewWindow description structure, tests the passed screen pointer to determine where you want your window opened, and modifies the Window and RastPort pointers in the calling function. The source code on disk contains a lot of documentation explaining the use of this and other functions in the Display.o module. You should take a good look it.

YOUR FIRST COMPLETE DISPLAY

What you now need is a program that uses all the things we have developed so far, providing you with a starting point for writing experimental programs. On disk is a the program called First_Display.c. I reproduce it below:

```
/* First_Display.c

   Paul Castonguay                                    July 15, 1991
   ========================================================== */

#include <intuition/intuition.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <proto/dos.h>

#include "Libs.h"
#include "Display.h"

VOID main(int argc, char *agrv[]);

VOID main(int argc, char *agrv[])
{
   struct Screen *my_screen;
   struct ViewPort *my_svp;
   struct Window *my_window;
   struct RastPort *my_rp;

   if(!Open_Libs())
   {
      printf("Trouble opening libraries\n");
      Delay(100);
      exit(RETURN_WARN);
   }

   if(!Open_My_Screen("LACEHIGH8", &my_screen, &my_svp))
   {
      Close_Libs();
      printf("Cannot open screen.\n");
      Delay(100);
      exit(RETURN_WARN);
   }

   if(!Open_Blank_Window(my_screen, NULL, &my_window, &my_rp))
   {
      CloseScreen(my_screen);
      Close_Libs();
      printf("Cannot open window.\n");
      Delay(100);
      exit(RETURN_WARN);
   }

   /* ================== Your code starts here =============== */

   SetRGB4(my_svp, 0, 3, 3, 3);
   SetAPen(my_rp, 5);
   Move(my_rp, 50, 50);
   Text(my_rp, "Help me!  I'm stuck in here.", 28);
   Delay(500);


   /* =============== Close shell before quitting ============ */

   if(my_window)
      CloseWindow(my_window);

   if(my_screen)
      CloseScreen(my_screen);

   Close_Libs();
}
```

This program opens a high resolution, 8 color, interlace screen and a full-size borderless window. If you prefer to program in a standard (non-interlace) screen simply change the "LACEHIGH8" resolution to "HIGH8". You will notice that I have inserted my usual silly comments in the window. That's the location where your program instructions could be inserted.

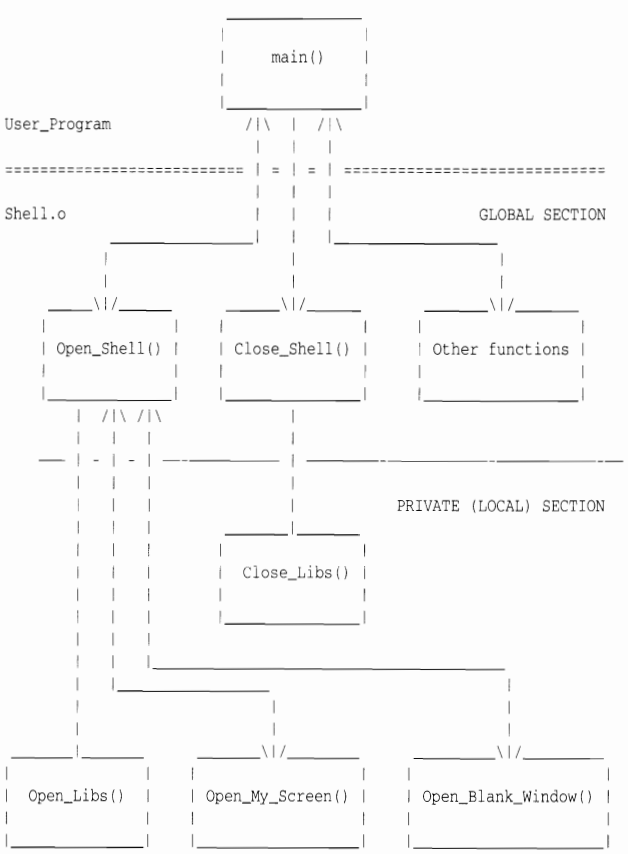
At this point I must say a few words about where I have brought you. All programmers, and authors, have their own preferences when it comes to using a windowing environment. Some want to hide all details and write absolutely transportable code, designing small interface functions to deal with the operating system whenever necessary. An application so designed could be transported to a completely different computer simply by redesigning those interface functions. The program proper would remain intact. That is a very admirable goal and I don't discourage it, although the extra overhead does slow down your application somewhat, not to mention the dedication required to write all those interface functions. At the other end of the spectrum are those who write programs in a linear fashion, using very few functions and lots of global variables. They argue that that is how to take maximum advantage of the operating system. I don't agree. Your programs will become so complex that you won't understand them. Unless you are extremely gifted, and I am not, you will probably give up long before you ever begin to take advantage of anything. The above program is in between. It is system dependant, declaring such things as window and screen pointers in main(), yet it is reasonably structured, simplifying the design of your programs. But what about those instructions to open a library, a screen, and then a window. Shouldn't they be combined into one call, making main() even more compact? Of course. In the next section I give you the equivalent of the above program, but with one extra level of structure. It essentially hides everything except the pointers that you will need to render graphics later. It also provides an excellent opportunity to learn another concept in structured programming.

ONE MORE LEVEL OF STRUCTURE

On disk is the module Shell.o and its related header file Shell.h. Together they are functionally equivalent to the combination of Libs.o, Libs.h, Display.h, and Display.h. However there is a structural difference that I will present in a minute. Along with Shell.o and Shell.h you will find a program called User_Program.c. This is supposed to represent any program that you may choose to write. It operates by making a single function call to Open_Shell(), passing to it all the pointers that you will need to render images and text. It also calls Close_Shell() before quitting.

```
/* User_Program.c

   Paul Castonguay                                    July 15, 1991
   ========================================================== */
#include <intuition/intuition.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <proto/intuition.h>
#include <proto/graphics.h>
#include <proto/dos.h>

#include "Shell.h"

VOID main(int argc, char *argv[]);

VOID main(int argc, char *argv[])
{
   struct Screen *my_screen;
   struct ViewPort *my_svp;
   struct Window *my_window;
   struct RastPort *my_rp;


   if(!Open_Shell(&my_screen, &my_svp, &my_window, &my_rp, "LACEHIGH8"))
   {
      printf("Problems in Open_Shell()\n");
      Delay(100);
      exit(RETURN_FAIL);
   }

   /* ================== Your code starts here =============== */


   SetRGB4(my_svp, 0, 3, 3, 3);
   SetAPen(my_rp, 5);
   Move(my_rp, 50, 50);
   Text(my_rp, "Help me!  I'm stuck in here.", 28);
   Delay(500);


   /* =============== Close shell before quitting ============ */

   Close_Shell(my_window, my_screen);

}
```

It is the job of Open_Shell() to call Open_Libs(), Open_My_Screen() and Open_Blank_Window() as you did before. Next, I give the hierarchical diagram.



There is more to this than the convenience of calling three functions in one line. The Open_Libs(), Open_My_Screen(), and Open_Blank_Window() functions themselves are hidden from User_Program.c, they are in a special private section. If you try to call them, you will get an unresolved function error from the linker. It is as if those functions did not exist. Yet there are other functions within Shell.o that you can call and we will be doing that in a minute.

You now see that structured programming is more than hiding variables in functions, it is also hiding functions in modules. Why do this? Because once you start getting used to opening your display by calling Open_Shell() you will soon forget that those other functions, Open_Libs(), Open_My_Screen() etc., exist. Out of sight, out of mind. The danger is that you may later design a new function and try to give it the same name as one already present in one of your modules. Worse, you may want to import an entire module consisting of many functions written by someone else that has a similar conflict. Hiding functions in modules has the same advantage as hiding variables in functions, it allows you to expand your project without fear of conflict.

To hide a function in a module, declare it as static (K&R, page 83) in both its prototype declaration as well as its definition.

→

DRAWING AT LAST

I now present a series of simple examples that demonstrate how to use the most fundamental graphic functions on the Amiga. They require that you already have a reasonable understanding of arrays, trigonometric functions, and loops.

Every window has a RastPort pointer that helps the system render graphics. To use graphics and text rendering functions, you must make reference to the RastPort pointer. You can look at the details of this structure in the <graphics/rastport.h> header file if you like, but you don't need to understand any of its members. In the above programming shell the RastPort pointer is assigned by Open_Shell() to the variable my_rp, which you must declare in main() (see above listing of User_Program.c).

The simplest drawing function is WritePixel(). It is equivalent to PSET in AmigaBASIC and PLOT POINTS in True BASIC. Here is an example:

```
WritePixel(my_rp, 100,100);
```

The first argument is the famous RastPort pointer, the second is the horizontal pixel number, measured from the left edge of the screen. The third is the vertical pixel number, measured from the top. The instruction illuminates a single pixel. To find out more about pixels and how their positions are measured on the Amiga, you should use AmigaBASIC.

The next function allows us to draw lines. Here is an example:

```
Draw(my_rp, 225, 75);
```

This causes the system to draw a straight line from the current pixel position to one addressed by 225,75. What is the current pixel position? It's the position of the last pixel that was drawn, or 0,0 if you have not yet issued any graphics functions. The above two instructions draw a straight line between 100,100 and 225,75.

To draw lines between any two points you use the Move() function to place the current pixel at the first endpoint of the line.

```
Move(my_rp, 50, 25);
Draw(my_rp, 275, 135);
```

PIXEL INDEPENDENT GRAPHICS (SCALING)

If you have had experience with a professional drawing environment, like True BASIC, you know that images are often drawn using mathematical equations written in Cartesian coordinates, not pixel numbers. Even without equations, it is usually easier to enter coordinate data that is related to the object being drawn, not some system of pixels that has nothing to do with it. Well, Intuition doesn't support that. The good news is that the Shell.o module of this article does. To do that, it uses a global structure called XY_Scale, a function called Scale_Window(), and two functions called Fx() and Fy(). Here is the template of XY_Scale as it appears in the Shell.c file:

```
/* === Global variable declarations === */

struct XY_Scale
{
   SHORT xpixels;
   SHORT ypixels;
   DOUBLE xmin;
   DOUBLE ymin;
   DOUBLE xmax;
   DOUBLE ymax;
   DOUBLE dx;
   DOUBLE dy;
};

struct XY_Scale s;
```

Here you see definitions for the left, right, bottom, and top edges of the window. You must assign these variables yourself. The other members are calculated and assigned for you by the Scale_Window() function in Shell.o. I'll show you an example in a minute.

Previously I scorned the indiscriminate use of global variables, but I never said you shouldn't use one when there was good reason to. It turns out that scaling can really slow down a graphics intensive program. Every graphics function needs to have its coordinates calculated in pixel numbers and keeping the above structure private to only those functions that use it would mean passing it an excessive number of times. It's not worth it! Better to declare it globally.

This is how you scale a window:

```
s.xmin = -3.1416;
s.xmax =  3.1416;
s.ymin = -1;
s.ymax =  1;

if(!(Scale_Window(my_window))
{
   print("Trouble scaling window\n");

   ... take corrective action ...
}
```

The Scale_Window() function measures the number of pixels on the screen and calculates dx and dy, the width and height of a single pixel. It then assigns these values to the global XY_Scale structure, from which all functions can use them. Note that ymin represents the bottom of the display, thus coordinates increase going up, as indeed they should.

To draw using Cartesian coordinates there are two pixel calculating functions, Fx() and Fy(). The above draw instruction using these functions, and the above scaling, could be written like this:

```
Draw(my_rp, Fx(-2.0), Fy(0.75));
```

Illuminating whatever pixel is at those Cartesian coordinates.

SINE

There is a program on disk called Sine.c that draws a single sine wave using the above window scaling. It demonstrates how s.xmin and x.xmax, and the calculated value of dx, can be used to plot a scientific equation on the screen. Here is the important part of that program:

```
s.xmin = -3.1416;
s.xmax =  3.1416;
s.ymin = -1.0;
s.ymax =  1.0;

if(!Scale_Window(my_window))
{
   Close_Shell(my_window, my_screen);
   exit(RETURN_WARN);
}

Move(my_rp, Fx(s.xmin), Fy(sin(s.xmin)));

for(x = s.xmin; x <= s.xmax+s.dx/2; x += s.dx)
   Draw(my_rp, Fx(x), Fy(sin(x)));

Delay(500);
```

The window is scaled for -3.1416 to 3.1416 radians horizontally, to display a single sine wave. Vertical scaling is for -1 and +1, the limits of the sine function. Initially the current pixel is moved to the first point on the curve, along the left edge of the screen. Then a FOR loop draws and increments across the screen in steps of dx, the width of a single pixel. The Fx() and Fy() functions are used to convert Cartesian coordinates to pixel numbers for the Draw() function. The dx/2 term is added to the loop's upper limit to prevent round off error caused by repetitively adding dx to x. Notice that reference to the exact number of pixels is not made in the above program. Pixels are a system dependant characteristic that you should not concern yourself with. The above program works independent of pixels. Try it out yourself! Change the resolution in the example on disk to LOW8. I'll say more about this next issue.

CIRCLE

The following example draws a circle using sine and cosine functions. I present this not as a recommended method of drawing circles in all your programs, but as background for future examples. Many interesting graphic objects require that you calculate points lying on the perimeter of a circle, as you will see in my last example.

```
s.xmin = -1.4;
s.xmax =  1.4;
s.ymin = -1.0;
s.ymax =  1.0;

if(!Scale_Window(my_window))
{
   Close_Shell(my_window, my_screen);
   exit(RETURN_WARN);
}

Move(my_rp, Fx(cos(Angle)), Fy(sin(Angle)));

for(Angle = 0.0; Angle <= 6.3; Angle += 0.1)
   Draw(my_rp, Fx(cos(Angle)), Fy(sin(Angle)));
```

The window is scaled to make the circle appear full size. Angle is incremented in steps of .1 radians while a line is drawn from point to point. See the example on disk.

Shell.o contains a function called Rad() which allows you to build your circle using degree measure.

```
Move(my_rp, Fx(cos(Rad((DOUBLE)Angle))), Fy(sin(Rad((DOUBLE)Angle))));

for(Angle = 0; Angle <= 360; Angle += 5)
   Draw(my_rp, Fx(cos(Rad((DOUBLE)Angle))), Fy(sin(Rad((DOUBLE)Angle))));
```

Observant readers will notice that I have changed the type of Angle from DOUBLE to SHORT and increased the step size of the loop. See the example on disk called Circle2.c To compile use the command:

```
LMK -f LMKFile_Circle2
```

At this point you might be wondering about the use of DOUBLE for floating point calculations. Isn't that slower than single precision? No! Refer to page G45 of your SAS/C documentation for an explanation.

COLORS

Changing colors is done with the SetRGB4() function. Colors on the Amiga are a property of screens, not windows, and for that reason you have to make reference to the ViewPort pointer, not the RastPort pointer. Here is an example:

```
SetRGB4(my_svp, 1, 15, 0, 0);
```

The first argument is the ViewPort pointer, the second is the color register number you want to change the color of, and following that are the three red-green-blue specifications of the color you want, 0 for minimum intensity, 15 for full intensity. The above instruction changes the color of color register 1 to bright red.

➔

PENS

Pens, or color registers, are chosen using the SetAPen() function.

```
SetAPen(my_rp, 7);
```

You use the RastPort pointer, not the ViewPort, to do this. Although colors are a property of screens, the system keeps track of which Pen it is using to draw with in each window. The above instruction changes the drawing pen to color register 7.

RANDOM NUMBERS

Random numbers are created using the rand() function of SAS/C. See page L168 of your documentation. To make its use a little easier I have designed a function which sets the initial seed to a number generated by the internal clock of the computer. That function is called Randomize() and it is in Shell.o. Here is an example that uses these features.

```
Randomize();

SetRGB4(my_svp, 0, 3, 3, 3);  /* set background grey */

for(i=0; i<1000; ++i)
{
   SetAPen(my_rp, rand()%8);
   WritePixel(my_rp, rand()%320, rand()%200);
}
```

Notice how I bring the random number within desired range using the modulus operator. See the example on disk called Spots.c

ROSETTE

A Rosette is an attractive image that is easy to build. To make one you divide the circumference of a circle into any number of equal parts, then draw lines between the endpoints of all the parts. The following example demonstrates the construction of a 23 point rosette:

```
s.xmin = -1.4;
s.xmax =  1.4;
s.ymin = -1.0;
s.ymax =  1.0;

if(!Scale_Window(my_window))
{
   Close_Shell(my_window, my_screen);
   exit(RETURN_WARN);
}

SetRGB4(my_svp, 0, 3, 3, 3);
SetAPen(my_rp, 5);

for(i = 0; i <=23; ++i)
{
   Rosette_Points[i][0] = Fx(cos(Rad(Angle)));
   Rosette_Points[i][1] = Fy(sin(Rad(Angle)));
   Angle += 360/23;
}

for(i = 0; i<=23; ++i)
   for(j=i; j<=23; ++j)
   {
      Move(my_rp, Rosette_Points[i][0], Rosette_Points[i][1]);
      Draw(my_rp, Rosette_Points[j][0], Rosette_Points[j][1]);
   }
```

First the window is scaled to draw a full-size circle. Next the background is set to grey and pen 5 is chosen for drawing. Then a FOR-LOOP calculates the endpoints of 23 equal segments along the circumference. This is done by calculating points at 23 equal angle positions, starting at 0 degrees. Horizontal coordinates of the points are cos(Rad(Angle)), vertical coordinates are sin(Rad(Angle)). It works just like in the construction of any circle except that here we calculate only 23 points. Coordinates are stored in a 23 by 2 array called Rosette_Points. Note the use of the Fx() and Fy() functions for these calculations. Thus the numbers stored in the array are Pixel numbers! Finally a double-nested FOR-LOOP draws lines between all points stored in the array. See the example on disk called Rosette.c.

Interesting variations can be achieved by changing the number of points (don't forget to re-dimension the array) or drawing between alternate points.

```
for(i = 0; i<=23; ++i)
   for(j=i; j<=23; j+=2)
   {
      Move(my_rp, Rosette_Points[i][0], Rosette_Points[i][1]);
      Draw(my_rp, Rosette_Points[j][0], Rosette_Points[j][1]);
   }
```

Here the loop variable j is incremented by 2. Try incrementing by 3 as well

WHAT NEXT

Today you have been able to get your feet wet drawing simple images. As you run the examples you may notice that some of them are unexpectedly slow. Isn't C supposed to be a fast language? Yes it is, but certain operations take time regardless of language. The above examples had to make a variety of floating point calculations, as well as a high number of system function calls. It turns out that many programs are fast not because of the language they are written in, but because of how they use it. Intuition has many high level (read easy to use) features that you can use to render images and text at faster speeds that what you have been seeing here. That will be the subject of the next article in this series.

Using Interrupts for Animated Pointers

by Jeff Lavin
The Puzzle Factory, Inc.

Programmers often create animated mouse pointers by breaking whatever the program is doing into small pieces, and having the program change the pointer's sprite image at each break, so as to give the impression of movement. This method has two big faults.

First, because the pieces of the task execute at different speeds, you can generally see the animation speeding up and slowing down as different sections of code are encountered. What's worse, animations vary even more when the program is run on different systems. The animation done by polling will obviously execute much slower on a vanilla A500 with a 7MHz 68000 than on an A3000 with a 25MHz 68030.

Second, and perhaps more important, is that the task might well execute more efficiently and in less time if it didn't have to keep stopping to do something else. This necessity to break up the program flow in order to update displays is the type of behavior we expect to see on non-multitasking systems, not on the Amiga.

In the program listed below, IntDemo.asm, we present a better way of keeping the pointer moving smoothly, and at the same time demonstrate the proper way to use two types of interrupts on the Amiga. To summarize, after opening libraries and our window, we call MakePointer(), which initializes both a soft interrupt and a vertical blank interrupt server. The server immediately begins counting vertical blanks. Each time it reaches 6 vertical blanks (1/ 10th second on NTSC systems,) it creates a soft interrupt that Cause()s another interrupt routine to increment the image used by this window's sprite. We use a second interrupt routine because we don't want to spend too much time in a vertical blank interrupt. There are eight 'clock' images used. This number represents a compromise between jerky animation and code size.

At this point, the 'clock' is running any time our window is active, and we never have to devote any program time or other resources, especially our attention while we are trying to do something else, to keeping the 'clock' running smoothly.

And run smoothly is exactly what it does! The salient feature of this approach to pointer animation is consistency. The 'clock' always runs at the same speed regardless of what the program is doing. This both looks better to the user, and is easier to program. It's like having your cake and eating it too. Since we are set up to count six vertical blank interrupts, and there are eight images, then 6 * .167 second * 8 = .8 seconds per cycle for NTSC users, and 6 * .02 second * 8 = .96 seconds per cycle for PAL users. This period is totally independent of processor speed, and independent of other tasks as well, as long as nothing is generating so many high priority interrupts that the Cause() interrupts get swamped. The period is easily changed by changing the appropriate maximum counts.

The executable can be run under V1.3 or V2.0 from CLI or Workbench. To run from the CLI, type:

```
1> [run] IntDemo
```

If run from the Workbench, just click on the icon. There are no arguments expected. To stop the demo, click on the window's close gadget. Of course, to see the 'clock' running, the demo window must be active. IntDemo can be run multiple times with no harm.

Note: All code used in this article is written in new M68000 family syntax. The new syntax was developed by Motorola specifically to support the addressing capabilities of the new generation of CPUs.

```
*****************************************************************************
* Program: IntDemo.asm
* Descrip: Makes the pointer into a running clock.
*          Runable from CLI or Workbench.
*   Usage: 1> [run] IntDemo or click on the icon.
*  Author: Jeff Lavin
* History: 06/26/91 V1.0 Created
*****************************************************************************

SysBase     equ     4                   ;aka AbsExecBase

SYS         macro                       ;Call a library function
            jsr     (_LVO\1,a6)
            endm
```

These are the equates for our work area. 'SO' means Structure Offset, and assembles much more quickly than the exec macros. Our work area uses the 'DX' data type, which takes no room on disk, but must be cleared by us.

```
WorkArea    clrso
_WBenchMsg  so.l    1                   ;Startup msg from Workbench
_GfxBase    so.l    1                   ;Ptr to graphics.library base
_IntBase    so.l    1                   ;Ptr to intuition.library base
_NewWindow  so.b    nw_SIZE             ;NewWindow structure
_WindowPtr  so.l    1                   ;Ptr to our demo window
_Seed       so.l    1                   ;Seed for random number generator
IntActive   so.w    1                   ;Interrupts have been installed
VBCounter   so.w    1                   ;Vertical blank interrupt counter
ImageIndex  so.w    1                   ;Index into table of image ptrs
VBInterrupt so.b    IS_SIZE             ;Vertical blank interrupt
SWInterrupt so.b    IS_SIZE             ;Soft interrupt
            alignso 0,4                 ;Align to next longword
Work_Size   soval

IntDemo     lea     (DT,pc),a5          ;Get our relative base
            movea.l a5,a0
            moveq   #(Work_Size/4)-1,d1 ;Subtract 1 for the dbcc
            moveq   #0,d0
..Loop      move.l  d0,(a0)+            ;Clear work area
            dbra    d1,..Loop

            movea.l (SysBase).w,a6
            movea.l (ThisTask,a6),a2    ;Our task base
            tst.l   (pr_CLI,a2)         ;From CLI or WBench?
            bne.b   .FromCLI

            lea     (pr_MsgPort,a2),a0
            SYS     WaitPort
            lea     (pr_MsgPort,a2),a0
            SYS     GetMsg              ;Get WBench startup msg
            move.l  d0,(_WBenchMsg,a5)

.FromCLI    lea     (GfxName,pc),a1     ;Open graphics.library
            moveq   #0,d0               ;Any version
            SYS     OpenLibrary
            move.l  d0,(_GfxBase,a5)
            beq.w   Cleanup

            lea     (IntName,pc),a1     ;Open intuition.library
            moveq   #0,d0               ;Any version
            SYS     OpenLibrary
            move.l  d0,(_IntBase,a5)
            beq.w   Cleanup
            movea.l d0,a6               ;IntuitionBase is in A6
```

Note that if we were opening a screen, instead of just a demo window, we might want to set the sprite colors so that our 'clock' always looks right no matter what the colors are on the user's system. Don't remove the comment markers and attempt to assemble - other code is needed.

```
;           movea.l (_ScreenPtr,a5),a2
;           lea     (sc_ViewPort,a2),a2 ;Get screen's ViewPort
;           movea.l a2,a0
;           moveq   #17,d0              ;Color register
;           moveq   #$F,d1              ;Red
;           moveq   #$0,d2              ;Green
;           moveq   #$0,d3              ;Blue
;           movea.l (_GfxBase,a5),a6
;           SYS     SetRGB4
;
;           movea.l a2,a0
;           moveq   #18,d0              ;The mouse pointer (sprite 0)
;           moveq   #$0,d1              ;uses these three color regs
;           moveq   #$0,d2
;           moveq   #$0,d3
;           SYS     SetRGB4
;
;           movea.l a2,a0               ;These particular colors are
;           moveq   #19,d0              ;the ones used for the 'clock'
;           moveq   #$E,d1              ;busy pointer under Workbench
;           moveq   #$E,d2              ;V2.0
;           moveq   #$C,d3
;           SYS     SetRGB4
```

We generate the NewWindow structure dynamically here so that we can center our window on the Workbench screen.

```
            suba.l  a1,a1               ;Screen
            moveq   #WBENCHSCREEN,d1    ;Type
            moveq   #sc_MouseY,d0       ;Size
            lea     (_NewWindow,a5),a2  ;Use handy buffer
            movea.l a2,a0
            SYS     GetScreenData       ;Get Workbench screen data
            tst.l   d0
            beq.w   Cleanup             ;Can't get Workbench screen data

            move.w  (sc_Width,a2),d2
            lsr.w   #1,d2               ;nw_Width = 1/2 of screen
            move.w  (sc_Height,a2),d3
            lsr.w   #1,d3               ;nw_Height = 1/2 of screen
            move.w  d2,d0
            lsr.w   #1,d0               ;nw_LeftEdge = 1/2 of remainder
            move.w  d3,d1
```

```
                lsr.w   #1,d1                   ;nw_TopEdge = 1/2 of remainder

                movea.l a2,a0                   ;NewWindow structure
                move.w  d0,(a2)+                ;nw_LeftEdge
                move.w  d1,(a2)+                ;nw_TopEdge
                move.w  d2,(a2)+                ;nw_Width
                move.w  d3,(a2)+                ;nw_Height
                moveq   #-1,d0
                move.w  d0,(a2)+                ;nw_DetailPen,nw_BlockPen
                move.l  #CLOSEWINDOW,(a2)+
                move.l
#WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE!ACTIVATE!SMART_REFRESH!NOCAREREFRESH,(a2)+
                lea     (WindowTitle,pc),a1
                move.l  a1,(nw_Title,a0)
                move.w  #WBENCHSCREEN,(nw_Type,a0)
                SYS     OpenWindow              ;Open our demo window
                move.l  d0,(_WindowPtr,a5)
                beq.w   Cleanup
```

Create our animated pointer. This is designed as a single function call for convenience. Just call it when you want the 'clock'.

```
                bsr.w   MakePointer
```

Here we just sit in a loop, drawing colored dots in the window. Between drawing dots we check to see if the user has hit the close gadget. We don't Wait() or WaitPort() because anytime we're not checking to see if there has been a message, we're busy drawing dots! On the other hand, because we're always executing code and never sleep, we're a cpu hog. Notice that there is absolutely no program code running the animated clock; it's completely handled by the interrupt code.

```
.GetMsg         movea.l (_WindowPtr,a5),a0      ;Ptr to our demo window
                movea.l (wd_UserPort,a0),a0
                movea.l (SysBase).w,a6
                SYS     GetMsg
                tst.l   d0                      ;Did we get an IntuiMsg?
                bne.b   .GotMsg                 ;Yes, check it out
                bsr.w   DrawDot                 ;No, draw a dot in our window
                bra.b   .GetMsg                 ;Check for a msg again

.GotMsg         movea.l d0,a1                   ;Ok, we've got an IntuiMsg
                move.l  (im_Class,a1),d2        ;Extract the IDCMP class
                SYS     ReplyMsg                ;Return the msg to Intuition
                cmpi.l  #CLOSEWINDOW,d2         ;Was it a CLOSEWINDOW msg?
                bne.b   .GetMsg                 ;No, keep looking for msgs
```

Remove our animated pointer. This is designed as a single function call for convenience. Just call it when you're done with the 'clock'.

```
                bsr.w   RemPointer
```

Pick up after ourselves, put away all of our toys, etc.

```
Cleanup         move.l  (_IntBase,a5),d0        ;Do we have a ptr to Intuition?
                beq.b   .CloseGfx               ;No, then we don't have a window
                movea.l d0,a6                   ;Yes, get ptr into A6
                move.l  (_WindowPtr,a5),d0
                beq.b   .CloseInt
                movea.l d0,a0
                SYS     CloseWindow             ;Close the demo window

.CloseInt       movea.l a6,a1
                movea.l (SysBase).w,a6
                SYS     CloseLibrary            ;Close intuition.library

.CloseGfx       movea.l (SysBase).w,a6
                move.l  (_GfxBase,a5),d0
                beq.b   .WBench
                movea.l d0,a1
                SYS     CloseLibrary            ;Close graphics.library

.WBench         move.l  (_WBenchMsg,a5),d2
                beq.b   .Exit
                SYS     Forbid                  ;So WBench won't UnloadSeg() us
                movea.l d2,a1
                SYS     ReplyMsg                ;Return startup msg
.Exit           moveq   #0,d0                   ;Always return OK
                rts

*****************************************************************************
* NAME:     MakePointer()
* FUNCTION: Initializes both a soft interrupt and a vblank interrupt server.
*           The interrupt server immediately begins counting vblanks.  Each
*           time it counts 6 vblanks (1/10th second in NTSC), it creates a
*           soft interrupt that Cause()s the soft interrupt server to
*           increment the image used by this window's sprite.
* INPUTS:   None
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************************

MakePointer     move.l  a6,-(sp)
```

This code checks to see if we have already been installed. Linking a second time would be fatal. It is not necessary to check in this example, but is included here for completeness.

```
                lea     (IntActive,a5),a0
                tst.w   (a0)                    ;Are we already active?
                bne.b   .AlreadyActive          ;Yes, exit
                addq.w  #1,(a0)                 ;No, show that we are active
```

We initialize the soft interrupt first, because it will be indirectly called by the VBlank interrupt, and must be ready to run immediately. Note that the soft interrupt doesn't need to be unlinked. When you're through with it, just don't Cause() it to be called anymore.

```
            lea     (SWInterrupt,a5),a1 ;Initialize soft interrupt
            move.b  #NT_INTERRUPT,(LN_TYPE,a1)
            move.b  #0,(LN_PRI,a1)
            lea     (SWIname,pc),a0         ;Our name
            move.l  a0,(LN_NAME,a1)
            move.l  a5,(IS_DATA,a1)         ;Ptr to our data
            lea     (MySWI,pc),a0
            move.l  a0,(IS_CODE,a1)         ;Interrupt routine to call
```

Now we can set up the VBlank interrupt. Since the code isn't critical, we just ask for a priority of zero.

```
            lea     (VBInterrupt,a5),a1     ;Initialize server interrupt
            move.b  #NT_INTERRUPT,(LN_TYPE,a1)
            move.b  #0,(LN_PRI,a1)
            lea     (ServerName,pc),a0      ;Our name
            move.l  a0,(LN_NAME,a1)
            move.l  a5,(IS_DATA,a1)         ;Ptr to our data
            lea     (MyServer,pc),a0
            move.l  a0,(IS_CODE,a1)         ;Interrupt server to call
            moveq   #INTB_VERTB,d0
            movea.l (SysBase).w,a6
            SYS     AddIntServer            ;Install that puppy
.AlreadyActive movea.l (sp)+,a6
            rts

*****************************************************************************
* NAME:     RemPointer()
* FUNCTION: Removes vblank interrupt server, and restores normal window
*           sprite.
* INPUTS:   None
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************************

RemPointer  move.l  a6,-(sp)
```

This code checks to see if we have already been removed. Attempting to unlink us a second time would be fatal. This check is not necessary in this example, but is included here for completeness.

```
            lea     (IntActive,a5),a0
            tst.w   (a0)                    ;Are we already active?
            beq.b   .NotActive              ;No, exit
            clr.w   (a0)                    ;Yes, show that we are inactive
```

We must remove the VBlank server first or as soon as we restore the window pointer, the soft interrupt will set it back. Since we are going to exit immediately, we don't care, but in your program, you might.

```
            lea     (VBInterrupt,a5),a1     ;Remove interrupt server
            moveq   #INTB_VERTB,d0
            movea.l (SysBase).w,a6
            SYS     RemIntServer
            movea.l (_WindowPtr,a5),a0
            movea.l (_IntBase,a5),a6
            SYS     ClearPointer            ;Restore window pointer
.NotActive  movea.l (sp)+,a6
            rts

*****************************************************************************
* NAME:     MyServer()
* FUNCTION: Vertical Blanking interrupt server.  Counts 6 vertical blanking
*           interrupts, or 1/10 second (in NTSC).  Note that a VBlank server
*           must preserve A0 if installed with a priority of 10 or higher.
* INPUTS:   A1 = Ptr to interrupt data area.
* RETURN:   Z = 1
* SCRATCH:  D0-D1/A1
*****************************************************************************

MyServer    move.l  a6,-(sp)
            move.w  (VBCounter,a1),d0       ;Get our current count
            addq.w  #1,d0                   ;Increment count
            cmpi.w  #5,d0                   ;Counted 6 vblanks yet?
            ble.b   .InBounds               ;No, keep counting
            moveq   #0,d0                   ;Yes, reset counter
.InBounds   move.w  d0,(VBCounter,a1)       ;Save new value
            bne.b   .NotYet                 ;Branch if less than 6 vblanks
            lea     (SWInterrupt,a1),a1
            movea.l (SysBase).w,a6
            SYS     Cause                   ;Signal our soft interrupt
.NotYet     movea.l (sp)+,a6
            moveq   #0,d0                   ;Tell handler to link next server
            rts

*****************************************************************************
* NAME:     MySWI()
* FUNCTION: Soft interrupt server.  Sets the window sprite to one of eight
*           'clock' images.
* INPUTS:   A1 = Ptr to interrupt data area.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************************

MySWI       movem.l d2-d3/a5-a6,-(sp)
            movea.l a1,a5
            movea.l (_WindowPtr,a5),a0      ;Ptr to our demo window
            lea     (ImageTable,pc),a1      ;Table of ptrs to images
            move.w  (ImageIndex,a5),d0      ;Index into above table
            lsl.w   #2,d0                   ;Make index for longworks
            movea.l (a1,d0.w),a1            ;Get current sprite image
            moveq   #16,d0                  ;Height
            moveq   #16,d1                  ;Width
            moveq   #-8,d2                  ;XOffset
            moveq   #-8,d3                  ;YOffset
            movea.l (_IntBase,a5),a6        ;Set sprite to new image
            SYS     SetPointer

.IncrPtr    move.w  (ImageIndex,a5),d0      ;Get index
            addq.w  #1,d0                   ;Increment index
            cmpi.w  #7,d0                   ;Counted 8 images yet?
            ble.b   .InBounds               ;No, keep counting
            moveq   #0,d0                   ;Yes, reset index
.InBounds   move.w  d0,(ImageIndex,a5)      ;Store new value
            movem.l (sp)+,d2-d3/a5-a6
            rts
```

```
ImageTable    dl      Clock0              ;Table of images
              dl      Clock1
              dl      Clock2
              dl      Clock3
              dl      Clock4
              dl      Clock5
              dl      Clock6
              dl      Clock7

*****************************************************************************
* NAME:     DrawDot()
* FUNCTION: Draws 'random' colored dots.
* INPUTS:   None
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************************

DrawDot       movem.l a2/a6,-(sp)
              movea.l (_WindowPtr,a5),a2
              movea.l (_GfxBase,a5),a6
              moveq   #49,d0
              bsr.w   Random              ;Get a new number
              andi.w  #3,d0
              movea.l (wd_RPort,a2),a1
              SYS     SetAPen             ;Set a new Front Pen color
```

Horizontal size of window - width of left border - width of right border

```
              move.w  (wd_Width,a2),d0    ;Get window height
              moveq   #0,d1
              add.b   (wd_BorderLeft,a2),d1  ;Get left & right borders
              add.b   (wd_BorderRight,a2),d1
              sub.w   d1,d0               ;Subtract borders
              bsr.w   Random
              sub.b   (wd_BorderRight,a2),d1  ;Add back right border
              add.w   d1,d0               ;Add left border to X
              move.l  d0,-(sp)            ;Save X
```

Vertical size of window - height of top border - height of bottom border

```
              move.w  (wd_Height,a2),d0   ;Get window height
              moveq   #0,d1
              add.b   (wd_BorderTop,a2),d1   ;Get top & bottom borders
              add.b   (wd_BorderBottom,a2),d1
              sub.w   d1,d0               ;Subtract borders
              bsr.w   Random
              sub.b   (wd_BorderBottom,a2),d1 ;Add back bottom border
              add.w   d0,d1               ;Add top border to Y
              move.l  (sp)+,d0            ;Retrieve X
              movea.l (wd_RPort,a2),a1
              SYS     WritePixel          ;Draw a dot
              movem.l (sp)+,a2/a6
              rts

*****************************************************************************
* NAME:     Random()
* FUNCTION: Generates psuedo-random numbers.
* INPUTS:   D0:16 = Range
* RETURN:   D0:16 = A number from 0 to (range-1).
* SCRATCH:  D0-D1/A0-A1
*****************************************************************************
```

This is a very inexpensive way to generate psuedo-random number sequences. However, it is sensitive to the input parameter when called repeatedly.

```
Random        move.l  d1,-(sp)
              lea     (_Seed,a5),a0       ;Get address of seed
              move.l  (a0),d1             ;Get seed
              add.l   d1,d1
              bhi.b   1$
              eori.l  #$1D872B41,d1
1$            move.l  d1,(a0)             ;Save new seed
              andi.l  #$FFFF,d1           ;Coerce into word
              divu.w  d0,d1               ;Divide by range
              swap    d1                  ; & get remainder (modulus)
              moveq   #0,d0
              move.w  d1,d0
              move.l  (sp)+,d1
              rts

GfxName       cstr    'graphics.library'
IntName       cstr    'intuition.library'
WindowTitle   cstr    'Interrupt Demo Window'
SWIname       cstr    'demo.swi.interrupt'
ServerName    cstr    'demo.vblank.server'
              even

DT            dx.b    Work_Size

              SECTION Images,DATA,CHIP    ;Put this data in CHIP ram

Clock0        dw      0,0
              dw      %0000010000000000,%0000011111000000
              dw      %0000000000000000,%0000011111000000
              dw      %0000000100000000,%0000011110000000
              dw      %0000000000000000,%0000011111000000
              dw      %0000011111000000,%0001111111111000
              dw      %0001111111110000,%0011111011111100
              dw      %0011111111111000,%0111111011111110
              dw      %0011111111111000,%0111111011111110
              dw      %0111111111111100,%1111111011111111
              dw      %0111111011111100,%1111111111111111
              dw      %0111111111111100,%1111111111111111
              dw      %0011111111111000,%0111111111111110
              dw      %0011111111111000,%0111111111111110
              dw      %0001111111110000,%0011111111111100
              dw      %0000011111000000,%0001111111111000
              dw      %0000000000000000,%0000011111100000
              dw      0,0

Clock1        dw      0,0
              dw      %0000010000000000,%0000011111000000
              dw      %0000000000000000,%0000011111000000
              dw      %0000000100000000,%0000011110000000
              dw      %0000000000000000,%0000011111100000
              dw      %0000011111000000,%0001111111111000
              dw      %0001111111110000,%0011111111111100
              dw      %0011111111111000,%0111111111011110
              dw      %0011111111111000,%0111111110111110
              dw      %0111111111111100,%1111111101111111
              dw      %0111111011111100,%1111111111111111
              dw      %0111111111111100,%1111111111111111
```

```
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock2  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0111111111111100,%1111111111111111
        dw      %0111111011111100,%1111111100000111
        dw      %0111111111111100,%1111111111111111
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock3  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0111111111111100,%1111111111111111
        dw      %0111111011111100,%1111111111111111
        dw      %0111111111111100,%1111111101111111
        dw      %0011111111111000,%0111111110111110
        dw      %0011111111111000,%0111111111011110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock4  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0111111111111100,%1111111111111111
        dw      %0111111011111100,%1111111111111111
        dw      %0111111111111100,%1111111011111111
        dw      %0011111111111000,%0111111011111110
        dw      %0011111111111000,%0111111011111110
        dw      %0001111111110000,%0011111011111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock5  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0111111111111100,%1111111111111111
        dw      %0111111011111100,%1111111111111111
        dw      %0111111111111100,%1111110111111111
        dw      %0011111111111000,%0111101111111110
        dw      %0011111111111000,%0111101111111110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock6  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0111111111111100,%1111111111111111
        dw      %0111111011111100,%1100000111111111
        dw      %0111111111111100,%1111111111111111
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

Clock7  dw      0,0
        dw      %0000010000000000,%0000011111000000
        dw      %0000000000000000,%0000011111000000
        dw      %0000000100000000,%0000001110000000
        dw      %0000000000000000,%0000011111100000
        dw      %0000011111000000,%0001111111111000
        dw      %0001111111110000,%0011111111111100
        dw      %0011111111111000,%0111101111111110
        dw      %0011111111111000,%0111101111111110
        dw      %0111111111111100,%1111110111111111
        dw      %0111111011111100,%1111111111111111
        dw      %0111111111111100,%1111111111111111
        dw      %0011111111111000,%0111111111111110
        dw      %0011111111111000,%0111111111111110
        dw      %0001111111110000,%0011111111111100
        dw      %0000011111000000,%0001111111111000
        dw      %0000000000000000,%0000011111100000
        dw      0,0

        end
```

List of Advertisers

Please use a Reader service card to contact those advertisers who have sparked your interest. Advertisers want to hear from you. This is the best way they have of determining the Amiga community's interests and needs. Take a moment now to contact the companies with products you want to lean more about. And, if you decide to contact a advertiser directly, please tell them you saw them in

AC's TECH/AMIGA

STOX

An ARexx-based System for Maintaining Stock Price History (using BaudBandit, Advantage and GEnie)

by Jack Fox

In this article I intend to describe a complete Amiga-based system for gathering and analyzing price and volume data on individual stocks. The major components of the system are The Advantage v1.1, BaudBandit v1.50, ARexx v1.10, Scripit v1.20, and AmigaDOS v1.3. The machine on which I run this is an A2000/030. I will describe the technical aspects in broad enough terms that some of the technical functions and principles can be applied to other applications, but this is not intended to be a tutorial on ARexx or any of the other software components.

A few years ago my (then computer-illiterate) father told me that he wanted to be able to maintain all his stock data on a computer by "just hitting one button." This got me to thinking that I could do it on my Amiga. At the time I had an A1000 without a hard disk, and while the computer was up to the task, I was soon to discover that the spreadsheet software available on the Amiga was inadequate. The entire system has evolved over the last year and a half through several releases of two different spreadsheet programs. I did not consider the system complete until v1.1 of The Advantage became available. This is because the system is designed around controlling the spreadsheet program through ARexx, and v1.1 of The Advantage is the only Amiga spreadsheet that I know of that is up to the task, although not the only spreadsheet that advertises ARexx compatibility.

My technical background is in IBM mainframe systems, so it was natural for me to approach this problem as a batch transaction processing system. By "batch" I mean a program or system of programs that is started and runs to conclusion without any user (the person behind the keyboard) intervention. A "transaction" can be thought of as a line or related lines of data that the program uses to perform a specific task. The task is repeated for each "transaction" until there are no more. There is also one part of the system which can be termed "real-time" and one which is "on-line".

The system requires that two software programs be active on your computer. One is the commercial program ARexx and the other is the freely distributable program Scripit. I initiate both from my Startup-Sequence as follows:

```
C:rexxmast
assign Xit: sys:Xit.v1.20
runback Xit:Scripit -x
```

I believe the "runback" was necessary to start Scripit under AmigaDOS v1.2 and get the Startup-Sequence CLI window to close afterward. It may not be necessary under v1.3 or later. The "-x" starts Scripit in the ARexx mode, so that it can receive commands from ARexx.

GATHERING THE DATA

A "real-time" application waits for input from some source outside of the program and responds with controlling commands. Controlling a space shuttle is a good example of a "real-time" application. The following is a very trivial example. Listing 1 shows the ARexx macro I use to download daily stock information from GEnie. (I shall use the term "macro" to describe a series of ARexx statements that are initiated from within another program, while an ARexx "program" is a series of statements that is executed on its own from the CLI.) Once logged on to GEnie all you have to do is navigate through the GEnie Star Services until you get to a prompt asking you to enter up to 30 comma separated stock ticker symbols. If the stock is tracked by GEnie, it will return data from the last two trading days in the following format:

```
 Description        Date     High     Low     Close  Volume
COMMODORE INTL LTD  01/14/91 10.2500  9.7500 10.0000 306800
                    01/11/91 10.7500 10.5000 10.5000 156400
```

(These lines are actually slightly shortened by removing spaces in order to fit onto one line of printed text.)

BaudBandit provides for associating a macro key with an ARexx macro. So once I have reached the ticker symbol prompt, I press F2 and the ARexx macro GEnieStox.rexx takes over. It takes about three minutes to download data for about 100 stocks. All I have to do after that is log off.

Let's take a look at how the macro works. The very first command is:

```
options results
```

This command is necessary any time you want ARexx to accept data from a "host" program. A "host" program is another program that ARexx recognizes and communicates with. ARexx will send any statements it does not recognize as ARexx commands to the "host" as commands. It is up to the "host" to determine if those are valid commands or not. But in order to accept any kind of response from the "host," "results" must be turned on. Since the macro GEnieStox.rexx was initiated from within BaudBandit, that is by default the "host" program. The next series of statements is all directed toward establishing the date in the format YYMMDD. This format is useful because it will sort correctly. In this case, they will be appended to the names of the files of captured data from GEnie to identify when the data was captured. One thing to be aware of however, is if data is recaptured on the same day, BaudBandit will add one to the date of the new files. Why? Because BaudBandit automatically increments a numeric appendage to a file name if the file already exists.

```
SaveDate = DATE('normal')
PARSE VAR SaveDate SaveDay SaveMonth SaveYear
SaveYear = RIGHT(SaveYear,2)
SaveMonth = UPPER(SaveMonth)
```

First the ARexx function "DATE" is invoked and the result is saved in the variable "SaveDate". One advantage of ARexx is the ability to define variables "on the fly." I could have just as well called "SaveDate" "TempWork", but "SaveDate" is more descriptive. The next three statements demonstrate ARexx's powerful string manipulation abilities. (A "string" is just a series of characters.) The "PARSE VAR" statement will divide the variable "SaveDate" into three new variables based on where spaces occur in "SaveDate". So "20 Feb 1990" becomes three new variables. The next command may be a little confusing. All that is happening is that the variable "SaveYear" is being manipulated by the "RIGHT" function to extract the right-most two characters. The results of

the function could have been saved in a new variable, but instead I am just re-using an existing variable, so that "1990" becomes "90". The "UPPER" function simply converts "SaveMonth " to all upper case. Once again the variable is re-used.

```
SELECT
  WHEN SaveMonth = 'JAN' THEN SaveMonthNum = '01'
...
```

The "SELECT" statement is used to assign numeric values to the months. Finally all the components of the date are assembled into a single variable:

```
FileDate = Saveyear||SaveMonthNum||SaveDay
```

and the first file of captured data is named:

```
CaptureFile = 'TEMP:GEnieStoxInput.'||FileDate
```

"TEMP:" must be a some directory previously ASSIGNed as the logical device "TEMP:". In our example here, "CaptureFile" will be given the value:

```
TEMP:GEnieStoxInput.900220
```

From here on the ARexx macro starts communicating with BaudBandit. The remaining lines are not recognized by ARexx, so they are sent to the host program.

```
Capt CaptureFile
```

opens the file named "TEMP:GEnieStoxInput.(date)". Remember we initiated this ARexx macro at a prompt for stock ticker

→

symbols. Now the macro sends a line of data to BaudBandit that tells it to send a string of data to GEnie followed by a carriage return:

```
Send 'acn,ko,cbu,...bmcs,dbrl\r'
```

What makes this macro a trivial example of "real-time" processing is that it responds to certain events in the output sent from GEnie. When it is done sending the data for the requested stock symbols, GEnie sends the following lines:

1. Retrieve Quote
2. Ticker Symbol Lookup

```
Enter # or <P>revious?
```

We tell BaudBandit to wait for the very end of this message:

```
Wait 'revious?'
```

and then have BaudBandit reply that we want to retrieve more quotes:

```
Send '1\r'
```

That is sending a "1" and a carriage return. GEnie responds by sending:

You may enter up to 30 ticker symbols, comma separated.
Enter Ticker Symbol(s) or <CR> to end?

And we send another series of ticker symbols.

As you can see in Listing 1, the data captured from GEnie is divided into three different files. The first file is data that is taken into the remaining components of the system. The second file is of ticker symbols that are not maintained by GEnie. If GEnie should start maintaining this data, I would like to know about it. The third file is of stock data that I am not yet maintaining in the rest of the system, but I want to collect the data in order to start tracking these stocks also.

EDITING THE DATA

In most cases one cannot be absolutely certain that the data coming into a software system is always valid. Therefore it is important to edit incoming data. Some of the things we want to protect our system from include:

1) the data unexpectedly appearing in a new format
2) duplicate data
3) miscellaneous errors in the data

Listing 2 is an ARexx program (StoxEditGEnie.rexx) to edit and reformat the data captured from GEnie. This is a batch program: once the program is initiated it reads through the input file until there are no more records. The following lines from the program illustrate the classic structure of a batch transaction processing program.

```
Instring = READLN('InFile')
LineCount = 1

DO WHILE ~ EOF('InFile')
  CALL EDITPASS
END
```

First an "initializing" read is performed to fetch the first line of input data. A counter is initialized to indicate that the first line of data has been read. This counter will be incremented each time a line of data is read. That way when errors are reported, the line number on which the error occurs can also be reported. Then we enter the main processing loop, where the function "EDITPASS" is repeatedly performed until we come to the end of file (EOF). Near the end of "EDITPASS" or a subfunction called within it, the next read of input data is performed. Each time we return to the start of "EDITPASS" ARexx determines whether or not we have come to the end of the file. Note that if the file were empty to begin with, "EDITPASS" would never be performed. A subsequent loop sorts the formatted data that has been written out, counts the entries for each stock, and checks for duplicate data.

Listing 3, StoxEdit.rexx, also performs editing and reformatting functions, except against manually entered data in a different format. In the history of my system, this was the original editing program before I started capturing data on GEnie. It is still useful for the few stocks that GEnie does not track. Since the data edited by this program was manually entered, the edits are more comprehensive. Be aware that this program changes the system date in order to edit whether entered dates are on the weekend, when of course stocks are not traded. The program does restore the proper system date, but a catastrophic event could leave you with a changed system date.

Error handling in both edit programs is handled by a common error handling function, ErrorReport.rexx, in Listing 4. The arguments to the error reporting function are a numeric error type (ErrorType), the line number of the input file in which the calling program has found the error (LineCount), the input line in error (InString), and the stock symbol related to the input line in error (Symbol). It formats an error message, writes the message and the input line in error to the console, and returns the error message (OutString) to the calling program. The following lines illustrate the common interface to this function.

```
OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
CALL WRITELN('MsgFile',OutString)
CALL WRITELN('MsgFile',InString)
```

Note that the error message and the input line in error are written to a message file from within each edit program. Although the error function automatically writes both to the console, it is also useful to write the messages to a file for later retrieval, since messages can scroll off the screen and be lost.

The program StoxEdit.rexx writes a file of formatted stock data called T:Stox. The input file name is the only argument to StoxEdit.rexx. To execute it from the CLI you would enter:

```
rx StoxEdit T:InputData
```

StoxEditGEnie.rexx writes a file in the same format as StoxEdit.rexx with the date of the input file appended to the name. The arguments to StoxEditGEnie.rexx are the number of days of data to use from the input file (1 being the most recent day, 2 being both days) and the date of the input file in the assigned path TEMP:. To execute it from the CLI you would enter:

```
rx StoxEditGEnie 2 910124
```

Finally, you may want to update your stock spreadsheets with input from more than one file. To do this, use the AmigaDOS program "Join" to combine the files into one file and then run it through the ARexx program in Listing 5, TotalPass.rexx:

```
join Stox.910122 Stox.910124 Stox as T:temp
rx TotalPass T:temp
```

TotalPass.rexx performs the functions of sorting input, counting the number of entries for each stock, and editing for duplicate data that is found at the end of both programs StoxEdit.rexx and StoxEditGEnie.rexx. The output file is named T:Stox and becomes input to the spreadsheet update program.

STARTING THE SPREADSHEET PROGRAM

The Advantage spreadsheet comes with a handful of native ARexx commands. All other ARexx interfaces to the spreadsheet must be done through executing a The Advantage macro. A macro is simply a command or series of commands that have been saved and can be re-executed. Consult the program manual for creating macros. The following macros must be created and saved to the directory "DH0:Advantage/Macros".

Macro	The Advantage Commands
calc	Commands,Recalculate
fill_down	Edit,Fill,Down
formula_to_value	Commands,Formula to Value
freeze_title	Options,Freeze,Row
paste_row	Edit,External,Paste File (enter "T:" as the path and "PasteFile" as the file and select "OK)
select_all	Edit,Select All
copy	Edit,Copy
paste	Edit,Paste,Relative
delete_column	Edit,Delete,Column

There are also five chart macros that need to be created, but I will explain them later.

Listing 6, StartAdv.rexx, is an ARexx program that starts The Advantage in interlace mode and loads the macros into the the spreadsheet program. From the CLI type:

```
rx StartAdv
```

Notice that the program was initiated from the CLI and not from within a "host" program. Therefore to communicate with any active programs, they must be addressed by the ARexx program.

```
ADDRESS XIT
'WAIT 250'
```

First the Scripit program is addressed and a command passed to it to wait five seconds while The Advantage program loads. (Remember, my system is an A2500/030 with a hard disk. Other systems may need to wait longer for the program to load.) Once loaded, The Advantage program is addressed and the "results" option turned on.

```
ADDRESS 'Advantage'
OPTIONS results
```

We then assign the full path and name of a macro to the variable "Macro" and execute The Advantage ARexx command "LoadMacros" using the variable "Macro" as the argument.

```
Macro = 'DH0:Advantage/Macros/calc'
'LoadMacros'
   Macro
```

➔

Finally, before we can begin building spreadsheets, we must construct a "model" spreadsheet that contains the column headings for our final spreadsheets. When a new spreadsheet is built for the first time, the model spreadsheet is used as the basis. In rows one and two enter and center the following labels in the columns noted:

Column	Label
A	DATE
B	HIGH
C	LOW
D	CLOSE
E	20 DAY AVERAGE
F	50 DAY AVERAGE
G	VOLUME
H	UP VOLUME
I	DOWN VOLUME
J	5 DAY UP/DOWN
K	20 DAY UP/DOWN
L	50 DAY UP/DOWN
M	50 DAY AVG VOLUME
N	AVG VOL %
O	NOTES

Row four should be filled with zeros in columns A through O. (You may notice that row three is not used. This is a result of the evolution of the system, and not for any good reason.) Under "Options, Preferences" select calculate by row and manual calculate, max rows of 500, max cols 100, and iterations of 1. Then save this as "model.adv" in a directory assigned to "STOX:".

CREATING AND UPDATING THE SPREADSHEETS

Finally we come to the heart of the system, that is creating and updating stock spreadsheets. Listing 7 is the ARexx program StoxUpdate.rexx. It assumes that a directory path called "STOX:" exists which contains the archived spreadsheets and the model spreadsheet (not archived.) It also assumes that the input data exists in file "T:stox". To start the program enter in the CLI

```
rx StoxUpdate
```

New spreadsheets will be created using the model spreadsheet and plugging in data from the input file and formulas defined within StoxUpdate.rexx. Existing archived spreadsheets will first be saved to an assigned directory path call "TEMP:" and then have new data added to them. Saving a spreadsheet before it is altered provides a back-up in case something unexpected occurs. A log file is maintained to record each stock that has been processed and the dates processed for that stock. An indicator file is maintained to record stocks which close within 15% of their recorded high close or which exhibit a surge in upside volume. Another file is maintained as a warning message file to record stocks whose high and low is more than 15% out of line or whose closing price is more than 15% out of line with the previous close. These factors could indicate faulty data or that the stock has split.

About the only limit to the number of stocks that can be maintained is disk space. The way the system currently works, the spreadsheets are stored in ZOOed format. In the history of the system I originally stored the programs in ARCed format, but as the size of the spreadsheets grew, the ARC program I had proved itself too slow. Even using ZOO, by far most of the processing time is spent in archiving and unarchiving the spreadsheets, approaching one minute for each 425 row spreadsheet on my system, while the rest of the process takes less than five seconds for adding one row to one spreadsheet.

Since some of the formulas used in the spreadsheets deal with fifty day averages, at least fifty days of data must be in the input file for a stock before a new spreadsheet is created. Once created, any number of days of data can be added to a spreadsheet.

The program reads all the input for a single stock, counting the number of input lines, and saves it off to a workfile. This way it knows how many rows to add to an existing spreadsheet or how many to create in a new one. For an existing spreadsheet, it loads the new data, "fills down" the formulas in the last row for as many rows as are added, recalculates the spreadhseet, and then converts all the formulas to values except for the last row. This saves space, since a value takes less storage than a formula, but it leaves the last row's formulas so that they can be replicated the next time the spreadsheet is updated. Finally the the spreadsheet is saved to "T:", which is used as a workspace, the spreadsheet is ZOOed, and then saved back to "STOX:". Finally the workspace is cleaned up and the next stock processed.

Listing 8 is an AmigaDOS script, StoxScript, used in the cleanup process. It is launched as a separate task from

StoxUpdate.rexx to copy the ZOOed spreadsheet back to "STOX:" and then delete it from "T:".

PROCESSING STOCK SPLITS

As any stock investor knows, companies sometimes declare a "stock split", that is all stockholders are issued a number of shares depending on how many they already hold. For instance in a two for one split, every stockholder is issued an additional share for each share held. In order to make sense out of the price history previous to a split, it must be adjusted in the same ratio as the stock split, since on the day of a stock split the market price typically is adjusted in the ratio of the split.

The program Stoxsplit.rexx in Listing 9 provides a way to adjust prices for a stock split. As with all the programs in this system that make use of The Advantage spreadsheet, it must have already been started using the program StartAdv.rexx. For instance if Commodore were to declare a three for two split, and it occurred on row 238 of your spreadsheet, you would enter the following on the CLI:

```
rx StoxSplit CBU 3/2 238
```

The program creates three work columns within the spreadsheet to adjust the high, low, and close prices previous to the split, then copies the new values back into the high, low, and close columns. The work columns are deleted. The moving averages of the closing price are recalculated, and finally the program determines a new all time recorded high and low closing price. A message is inserted in the "notes" column in the row in which the split occured indicating what kind of split took place.

GRAPHING THE DATA

There are two outputs to this whole system that I consider to be the really useful parts. The first is the indicator file produced by StoxUpdate.rexx. The second is the graphing capability of the program Open.rexx in Listing 10. (I mentioned earlier that portions of this system exhibited characteristics of batch, online, and real-time processing. This is the "online" portion of the system, although such designations are certainly open to debate.)

Once again, The Advantage spreadsheet program must be active and have been started by StartAdv.rexx. Five chart macros must exist, they are:

chart_1, a high-Low chart of high, low, and close

chart_2, a line chart of close, twenty, and fifty day moving average of closing price

chart_3, a floating bar chart of percent of day's volume above or below the fifty day moving average of the volume

chart_4, a line chart of the twenty and fifty day moving average of the ratio between volume on "up" days and "down" days

chart_5, a line chart of the five day moving average of the ratio between volume on "up" days and "down" days

The Advantage was started in interlace mode so that four charts can be viewed at once when they are all shrunk to their smallest vertical size and relocated on the screen. A fifth chart is stacked behind one of the viewed charts. They all have the same horizontal scale, therefore they all share the same time line. This chart shuffling is accomplished with Scripit, which is able to manipulate windows created by The Advantage. Unfortunately, all chart windows created by The Advantage have the same name as the spreadsheet window from which they were created. This causes a problem for Scripit, which addresses windows by their names. The program Open.rexx overcomes this problem by resizing and moving a chart window right after it has been created by The Advantage. To illustrate this, first The Advantage is addressed by the ARexx program

```
ADDRESS 'Advantage'
```

then the range to be graphed is selected

```
TopCorner = 'J'||RowStart
BottomCorner ='J'||LastRow
'SelectRange'
 TopCorner
 BottomCorner
```

the chart macro is run

```
macro = 'chart_5'
'macro'
 macro
```

then Scripit is addressed by its ARexx port name

```
ADDRESS XIT
```

Scripit is told to wait

```
'WAIT 5'
```

then the chart window is selected on The Advantage screen and activated

```
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
```

the window is resized

```
'W RESIZETO 640,120'
```

Scripit waits again

```
'WAIT 5'
```

and the window is repositioned.

```
'W MOVE 0 300'
```

The "waits" that Scripit executes are necessary for the last instruction to fully execute and for the system to be ready for the next instruction. Some systems may need a longer waiting period.

To graph data for a stock enter the stock symbol and the number of most recent days to graph. If no days are entered, the stock will be graphed for all the days present in the spreadsheet. For instance to graph the most recent 200 days for Commodore, enter in the CLI:

```
rx Open CBU 200
```

The program will close any chart windows from a previous execution of the program using Scripit. Once again, it is a little tricky because all the windows share the same name. So if you had manually shuffled the windows you may end up closing the spreadsheet window, which will make it impossible to open the new spreadsheet. If this happens you will have to open a new window before re-running Open.rexx. Open.rexx keeps track of the last stock graphed in an ARexx "clips" variable called "Last". You may at times have to clear out this variable by entering in the CLI:

```
rxset Last
```

CONCLUSION

I did not quite achieve my father's "one-button" dream system. It takes a few keystrokes to navigate through GEnie to the point where the ARexx macro captures data and a few keystrokes are needed to edit/reformat the data and consolidate data from multiple input files. One area of potential improvement is to build a "front-end" to the editing process. Another improvement that I will most likely implement is to remove the archiving of the spreadsheets from the entire system. It simply takes too much time in the updating process. While for monitoring only a few stocks it might be worthwhile, I reckon that hundreds of stocks could be maintained in less than half an hour without the archiving, while it would take hours with the archiving. A limit needs to be built into the program StoxUpdate.rexx as to the number of rows maintained, deleting rows at the beginning as new rows are added. I think 500 would be a good limit. That would be about two years of daily data. Lastly, it is a good idea to archive the final input data to StoxUpdate.rexx, so that the spreadsheets can be recreated if they are ever lost or need to be altered.

The Gold Disk Office product contains two programs, Calc and Graph, which are the spreadsheet and graphing capabilities of The Advantage split into two separate programs. I have never used them, but since Calc has the macro and ARexx capabilities of The Advantage, the spreadsheet maintenance portion of the Stox system should work. Unfortunately, Graph does not have a macro capability, so graphing would have to be done manually, unless a Scripit interface could be written.

I intentionally did not discuss the formulas used in the stock spreadsheets, since that is beyond the scope of this article. If you are interested in pursuing the technical analysis of stock data, I suggest William J. O'Neil's book, "How to Make Money in Stocks."

I hope this article shows that the Amiga is capable of serious business applications. The ability of a spreadsheet to be controlled by ARexx opens the possibility of constructing sophisticated applications. In the future, I look forward to the development of sophisticated ARexx applications interfacing to databases and expert system shells.

Listing One

```
/*********************************************************/
/*                    GEnieStox                          */
/*                                                       */
/* This macro is executed from BaudBandit while logged on */
/* to GEnie at the "Enter Ticker Symbol(s) or <CR> to     */
/*                 end?" prompt.                         */
/* It will save the stock data sent from GEnie to the     */
/* specified files.                                      */
/*                                        Jack Fox       */
/*********************************************************/

options results

/* Get today's date for appending to file names.         */
SaveDate = DATE('normal')
PARSE VAR SaveDate SaveDay SaveMonth SaveYear
SaveYear = RIGHT(SaveYear,2)
SaveMonth = UPPER(SaveMonth)

SELECT
  WHEN SaveMonth = 'JAN' THEN SaveMonthNum = '01'
  WHEN SaveMonth = 'FEB' THEN SaveMonthNum = '02'
  WHEN SaveMonth = 'MAR' THEN SaveMonthNum = '03'
  WHEN SaveMonth = 'APR' THEN SaveMonthNum = '04'
  WHEN SaveMonth = 'MAU' THEN SaveMonthNum = '05'
  WHEN SaveMonth = 'JUN' THEN SaveMonthNum = '06'
  WHEN SaveMonth = 'JUL' THEN SaveMonthNum = '07'
  WHEN SaveMonth = 'AUG' THEN SaveMonthNum = '08'
  WHEN SaveMonth = 'SEP' THEN SaveMonthNum = '09'
  WHEN SaveMonth = 'OCT' THEN SaveMonthNum = '10'
  WHEN SaveMonth = 'NOV' THEN SaveMonthNum = '11'
  WHEN SaveMonth = 'DEC' THEN SaveMonthNum = '12'
OTHERWISE
  NOP
END

FileDate = Saveyear||SaveMonthNum||SaveDay
CaptureFile = 'TEMP:GEnieStoxInput.'||FileDate
/* Start capturing anyting written to the screen.        */
Capt CaptureFile

/* Send a list of stock symbols to GEnie.                */
/* ARexx will consider everthing between the single quotes*/
/* as being one line.                                    */
Send 'ACN,KO,CBU,CMY,CPQ,CTB,DIS,FNM,KBH,LTR,MKC,MTC,NKE,
PHM,PD,MO,SCH,SRR,TDM,TKA,UT,APBI,ARBR,ACAD,BHAGA,BMCS,
DBRL,INTV,LIZC\r'

/* Wait until GEnie is done sending data and has displayed*/
/* a prompt of what to do next.                          */
Wait 'revious?'
```

```
/* Select option "1".                                          */
Send '1\r'

/* Wait for the prompt to send stock symbols.                  */
Wait ' to end?'

/* Send another list of symbols.                               */
Send 'MCIC,MNPI,NIIS,ORCL,QNTM,ROST,SHOR,WSGC,BLD,BER,TO\r'

/* Wait until GEnie is done sending data and has displayed*/
/* a prompt of what to do next.                                */
Wait 'revious?'

/* Select option "1".                                          */
Send '1\r'

/* Close capture file.                                         */
Capt Off

/* Name a new capture file, start capturing data to it,        */
/* and send new stock symbols to GEnie.                        */
CaptureFile2 = 'TEMP:GEnieTemp.'||FileDate
Capt CaptureFile2

Wait ' to end?'
Send 'CDNC,GNI,NWNL,SIVB,TBSA,TBSB,ZION\r'

Wait 'revious?'
Send '1\r'

Capt Off

CaptureFile3 = 'TEMP:GEnieStoxInput2.'||FileDate
Capt CaptureFile3

Wait ' to end?'
Send 'ABF,ADCT,AIP,APCC,ARG,ATM,AMD,ATVC,BA,BCO,BGSS,BMD,
BMET,BOCB,BORL,CBRL,CGNX,CNK,CNTX,CNX,CORD,CS,CSCO,CSIM,
DCI,DRYR,FAST,HCSG,HRC\r'

Wait 'revious?'
Send '1\r'

Wait ' to end?'
Send 'HTLD,INTC,INEL,KES,LDDSA,LFT,LIQB,LSCC,LSCP,MEDC,
MGMA,MGR,MGRC,MSII,NECC,NSH,NVCR,OEA,PHSY,PMTC,PWN,PYRD,
SBA,SCAF,SCRP,SMLS,SODA,SPLS,SSAX\r'

Wait 'revious?'
Send '1\r'

Wait ' to end?'
Send 'STSN,USS,VALN,VRES,WAB\r'

Wait 'revious?'
Send '\r'

Capt Off
```

Listing Two

```
/*************************************************************/
/*                  StoxEditGEnie                            */
/*                                                           */
/* This program reformats the data collected from GEnie      */
/* for use by StoxUpdate. Arguments to this program are      */
/* the number of days of data to be retrieved from the       */
/* file, and the date of the file to be edited in YYMMDD     */
/* format.                                                   */
/*                                      Jack Fox             */
/*************************************************************/

Arg NumberOfDays FileDate

FileDate = STRIP(FileDate)
InYear = LEFT(FileDate,2)

IF NumberOfDays = 1
THEN
   NOP
ELSE
   IF NumberOfDays = 2
   THEN
      NOP
   ELSE
      DO
      SAY 'number of days must be 1 or 2'
      CALL EOJ
      END

FileName = 'TEMP:GEnieStoxInput.'||FileDate

IF EXISTS(FileName)
THEN
   SAY 'using file '||FileName
ELSE
   DO
   SAY 'file '||FileName||' does not exist'
   CALL EOJ
   END

CALL OPEN('InFile',Filename,'read')
CALL OPEN('MsgFile','T:StoxEdit.msg','write')
CALL OPEN('SortIn','T:SortIn','write')
Instring = READLN('InFile')
LineCount = 1


DO WHILE ~ EOF('InFile')
   CALL EDITPASS
END

CALL CLOSE('SortIn')
ADDRESS COMMAND 'SORT T:SortIn T:Stox.'||FileDate
CALL OPEN('SortOut','T:Stox.'||FileDate,'read')

InString=READLN('SortOut')
LineCount = 1
StockEntryCount = 1
PrevSymbol = ' '

DO WHILE ~ EOF('SortOut')
   CALL TOTALPASS
END

IF StoxPresent = Y
   THEN
   DO
   OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
   CALL WRITELN('MsgFile',OutString)
   CALL WRITELN(StdOut,OutString)
   END
ELSE
   NOP


EOJ:

CALL CLOSE('MsgFile')
SAY 'Job Ended'
EXIT


EDITPASS:
/*Pass through the file one time to do edits*/


PARSE VAR InString FirstVar . . .

IF FirstVar = 'Tick'
   THEN
   CALL EDIT_INPUT
ELSE
   DO
   IF FirstVar = 'Ticker'
   THEN
      DO
      OutString = InString
      CALL WRITELN('MsgFile',OutString)
      CALL WRITELN(StdOut,OutString)
      END
   ELSE
      NOP
   END

InString = READLN('InFile')
Linecount = Linecount + 1

RETURN


EDIT_INPUT:
/* Since the data captured from GEnie does not have the   */
/* stock symbol, the symbol must be identified for each   */
/* name returned from GEnie. All stocks captured from     */
/* GEnie must be included in the SELECT statement below.  */

InString = READLN('InFile')
Linecount = Linecount + 1

TempSymbol = LEFT(InString,5)
TempSymbol = STRIP(TempSymbol)

TP = SUBSTR(InString,33,2)

Class = SUBSTR(InString,36,1)

IF TP = '  '
THEN
   CALL EDIT_LINES
ELSE
```

➔

```
    NOP

RETURN


EDIT_LINES:
/**/

IF NumberOfDays = 1
THEN
   DO
   CALL EDIT_LINE
   CALL WRITE_LINE
   END
ELSE
   DO
   CALL EDIT_LINE
   TempDate = OutDate
   CALL WRITE_LINE
   CALL EDIT_LINE
   IF TempDate = OutDate
   THEN
      DO
      OutString = ,
Symbol||' has duplicate days for '||OutDate
     CALL WRITELN('MsgFile',OutString)
     CALL WRITELN(StdOut,OutString)
      END
   ELSE
      CALL WRITE_LINE
   END

RETURN


EDIT_LINE:
/* Perform edits on the data on a line of input data.     */

InfoLine = SUBSTR(InString,38)

PARSE VAR InfoLine Var1 Var2 Var3 Var4 Var5
InDate = STRIP(Var1)
High = STRIP(Var2)
Low = STRIP(Var3)
Close = STRIP(Var4)
Volume = STRIP(Var5)

PARSE VAR InDate InMonth '/' InDay

TempVar = InMonth
CALL EDIT_NUMERIC

TempVar = InDay
CALL EDIT_NUMERIC

CALL EDIT_DATE

OutDate = InYear||InMonth||InDay

NotTraded = N

IF High = 'NOT'
THEN
    DO
    NotTraded = Y
    InString = READLN('InFile')
    Linecount = Linecount + 1
    RETURN
    END
ELSE
    NOP

TempVar = High
CALL EDIT_NUMERIC

TempVar = Low
CALL EDIT_NUMERIC

TempVar = Close
CALL EDIT_NUMERIC

TempVar = Volume
CALL EDIT_NUMERIC

CALL EDIT_WARN

IF Volume = 0
THEN
   DO
   OutString = Symbol||' has 0 volume on '||OutDate
   CALL WRITELN('MsgFile',OutString)
   CALL WRITELN(StdOut,OutString)
   END
ELSE
   NOP

InString = READLN('InFile')
Linecount = Linecount + 1

RETURN


EDIT_DATE:
/* Date Edits: valid date                                  */
/*           not a Saturday or Sunday                      */
/* Dates from GEnie should be valid trading dates, but     */
/* just in case...                                         */

ErrorType = 0

SELECT
   WHEN InMonth = '01' THEN CharMonth = 'JAN'
   WHEN InMonth = '02' THEN CharMonth = 'FEB'
   WHEN InMonth = '03' THEN CharMonth = 'MAR'
   WHEN InMonth = '04' THEN CharMonth = 'APR'
   WHEN InMonth = '05' THEN CharMonth = 'MAY'
   WHEN InMonth = '06' THEN CharMonth = 'JUN'
   WHEN InMonth = '07' THEN CharMonth = 'JUL'
   WHEN InMonth = '08' THEN CharMonth = 'AUG'
   WHEN InMonth = '09' THEN CharMonth = 'SEP'
   WHEN InMonth = '10' THEN CharMonth = 'OCT'
   WHEN InMonth = '11' THEN CharMonth = 'NOV'
   WHEN InMonth = '12' THEN CharMonth = 'DEC'
OTHERWISE Errortype = 5
END

IF ErrorType = 0
THEN
   NOP
ELSE
   DO
   OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
  CALL WRITELN('MsgFile',OutString)
  CALL WRITELN('MsgFile',InString)
   CALL EOJ
   END

RETURN


EDIT_NUMERIC:
/* Make sure numeric data is numeric.                      */

CALL DATATYPE(TempVar)
IF Result = NUM
THEN
   NOP
ELSE
   DO
   ErrorMsg = 'variable '||TempVar||' not numeric'
   OutString = '==> Line '||LineCount||' '||ErrorMsg
   CALL WRITELN('MsgFile',OutString)
   CALL WRITELN(StdOut,OutString)
   CALL WRITELN('MsgFile',InString)
   CALL WRITELN(StdOut,InString)
   CALL EOJ
   END

RETURN


EDIT_WARN:
/* If the difference between the high and low is greater  */
/* than 15%, there may be an error in the data, so issue  */
/* a warning message.                                     */

IF High > Low
   THEN
   DO
   Temp1 = High - Low
   Temp2 = Temp1 / High
   IF Temp2 > .15
      THEN
            DO
            ErrorType = 10
     OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
     CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
      END
   ELSE
      NOP
   END
ELSE
   NOP

RETURN


WRITE_LINE:
/* Put together an output line of data to go to sort      */
```

```
IF NotTraded = Y
THEN
    RETURN
ELSE
    NOP

IF Class = ' '
THEN
    NOP
ELSE
    DO
    wk = RIGHT(TempSymbol,1)
    IF Class = wk
    THEN
        NOP
    ELSE
        TempSymbol = TempSymbol||Class
    END

SymbolLength = (LENGTH(TempSymbol))
FirstVar = (UPPER(FirstVar))

IF SymbolLength = 1
THEN
   Symbol = TempSymbol||'     '
ELSE
   IF SymbolLength = 2
   THEN
      Symbol = TempSymbol||'    '
   ELSE
      IF SymbolLength = 3
      THEN
         Symbol = TempSymbol||'   '
            ELSE
               IF SymbolLength = 4
         THEN
            Symbol = TempSymbol||'  '
                  ELSE
            Symbol = TempSymbol||' '

OutString1 = Symbol||OutDate||' '||InDay||'-',
||CharMonth||'-'||InYear||' '
OutString2 = Volume||' '||High||' '||Low||' '||Close
OutString = OutString1||OutString2
CALL WRITELN('SortIn',OutString)

RETURN


TOTALPASS:
/* Get totals & dates for each stock and do final edit    */
/* for duplicates.                                        */

PARSE VAR InString Symbol SortDate Date Volume High Low ,
Close

IF PrevSymbol = Symbol
   THEN
   DO
   StockEntryCount = StockEntryCount + 1
   IF PrevSortDate = SortDate
      THEN
            DO
            ErrorType = 19
            OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
           CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
            END
   ELSE
      DO
      PrevSortDate = SortDate
            HoldLastDate = Date
            END
   END
ELSE
   IF StoxPresent = Y
      THEN
      DO
            OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
           CALL WRITELN('MsgFile',OutString)
     CALL WRITELN(StdOut,OutString)
            HoldFirstDate = Date
     HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END
   ELSE
      DO
            HoldFirstDate = Date
     HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END

StoxPresent = Y
InString = READLN('SortOut')
LineCount = LineCount + 1

RETURN
```

Listing Three

```
/*********************************************************/
/*                     StoxEdit                          */
/*                                                       */
/* This program takes an input file of stock activity,   */
/* Daily stock reports) edits the activity, and writes   */
/* a file of transactions for program StoxUpdate.        */
/*                                                       */
/* The input file consists of two types of lines, date   */
/* lines and stock activity lines. At least one date line */
/* of numeric format MMDDYY must be present, followed by */
/* as many activity lines as desired for that day:       */
/*                                                       */
/* 062990                                                */
/* acn 357 391/4 383/4 383/4                             */
/*                                                       */
/* Activity lines consist of 1) stock symbol, 4) volume, */
/* 5) high, 6) low, 7) close.                            */
/*                                                       */
/* The output file, which is input to program StoxUpdate */
/* is sorted by stock symbol and date so that all updates */
/* to a given spreadsheet take place together while that */
/* spreadsheet is only opened once. An output line looks */
/* like this:                                            */
/*                                                       */
/* ACN  900629 29-JUN-90 357 39.25 38.75 38.75           */
/*                                                       */
/* Error messages and counts of activity lines for each  */
/* individual stock are written both to a message file   */
/* and to the console.                                   */
/*                                        Jack Fox       */
/*********************************************************/

RC = 0
ErrorFlag = N
InYearPrev = 0
InMonthPrev = 0
InDayPrev =0


/* Input file name is parameter entered on command line. */
Arg Filename

IF Filename = ''
   THEN DO
   SAY 'Please enter input file name as parameter.'
   CALL EOJ
   END
ELSE
   IF EXISTS(Filename)
   THEN
      SAY 'using file '||Filename
   ELSE
      DO
      Filename = 'Ram:'||Filename
            IF EXISTS(Filename)
            THEN
               SAY 'using file '||filename
            ELSE
               DO
               Say Filename||' does not exist'
                     CALL EOJ
                     END
      END

/* Save off the current date for later date testing.     */

SaveDate = (DATE('normal'))
PARSE VAR SaveDate SaveDay SaveMonth SaveYear
SaveYear = (RIGHT(SaveYear,2))
SaveMonth = (UPPER(SaveMonth))

SELECT
  WHEN SaveMonth = 'JAN' THEN SaveMonthNum = '01'
  WHEN SaveMonth = 'FEB' THEN SaveMonthNum = '02'
  WHEN SaveMonth = 'MAR' THEN SaveMonthNum = '03'
  WHEN SaveMonth = 'APR' THEN SaveMonthNum = '04'
  WHEN SaveMonth = 'MAU' THEN SaveMonthNum = '05'
  WHEN SaveMonth = 'JUN' THEN SaveMonthNum = '06'
  WHEN SaveMonth = 'JUL' THEN SaveMonthNum = '07'
  WHEN SaveMonth = 'AUG' THEN SaveMonthNum = '08'
  WHEN SaveMonth = 'SEP' THEN SaveMonthNum = '09'
  WHEN SaveMonth = 'OCT' THEN SaveMonthNum = '10'
  WHEN SaveMonth = 'NOV' THEN SaveMonthNum = '11'
  WHEN SaveMonth = 'DEC' THEN SaveMonthNum = '12'
OTHERWISE
  NOP
END
```

```
CALL OPEN('InFile',Filename,'read')
CALL OPEN('MsgFile','T:StoxEdit.msg','write')
CALL OPEN('SortIn','T:SortIn','write')
Instring = READLN('InFile')
LineCount = 1


DO WHILE ~ EOF('InFile')
   CALL EDITPASS
END


CALL CLOSE('SortIn')
ADDRESS COMMAND 'SORT T:SortIn T:Stox'
CALL OPEN('SortOut','T:Stox','read')


InString=READLN('SortOut')
LineCount = 1
StockEntryCount = 1
PrevSymbol = ' '


DO WHILE ~ EOF('SortOut')
   CALL TOTALPASS
END


IF StoxPresent = Y
   THEN
   DO
   OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
  CALL WRITELN('MsgFile',OutString)
  CALL WRITELN(StdOut,OutString)
   END
ELSE
   NOP


EOJ:

IF ChangeDate = Y
   THEN
   ADDRESS COMMAND ,
'Date '||SaveDay||'-'||SaveMonth||'-'||SaveYear
ELSE
   NOP

CALL CLOSE('MsgFile')
SAY 'Job Ended'
EXIT


EDITPASS:
/*Pass through the file one time to do edits*/


PARSE VAR InString FirstVar Volume High Low Close

IF Volume = ''
   THEN
   CALL EDITDATE
ELSE
   DO
      CALL EDITLINE
            IF ErrorType = 0
               THEN
                     IF ErrorFlag = N
                        THEN
                              CALL WRITELINE
                     ELSE
                              NOP
            ELSE
               NOP
   END

IF ErrorType ~= 0
   THEN
      DO
      OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
     CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
            ErrorFlag = Y
      END
   ELSE
      NOP

Instring = READLN('InFile')
Linecount = Linecount + 1

RETURN
```

```
EDITDATE:
/* Date Edits: valid date                                    */
/*             not greater than current date                 */
/*             not a Saturday or Sunday                      */
/*             must be in ascending order                    */


InYear = (RIGHT(FirstVar,2))
Temp = (LEFT(FirstVar,4))
InDay = (RIGHT(Temp,2))
InMonth = (LEFT(Temp,2))

ErrorType = 0

CALL DATATYPE(FirstVar)

SELECT
   WHEN Result ~= 'NUM' THEN ErrorType = 1
   WHEN InYear < InYearPrev THEN ErrorType = 2
   OTHERWISE NOP
END

IF InYear = InYearPrev
   THEN
   IF InMonth < InMonthPrev
   THEN
      ErrorType = 2
   ELSE
      IF InMonth = InMonthPrev
            THEN
               IF InDay < InDayPrev
                        THEN
            ErrorType = 2
          ELSE
                        IF InDay = InDayPrev
                              THEN
                                 ErrorType = 3
                              ELSE
                                 NOP
      ELSE
               NOP
ELSE
   NOP


IF InYear > SaveYear
   THEN
   Errortype = 4
ELSE
   IF InYear = SaveYear
   THEN
      IF InMonth > SaveMonthNum
      THEN
         ErrorType = 4
      ELSE
         IF InMonth = SaveMonthNum
               THEN
                  IF InDay > SaveDay
            THEN
               ErrorType = 4
            ELSE
               NOP
          ELSE
                  NOP
   ELSE
      NOP


SELECT
   WHEN InMonth = '01' THEN CharMonth = 'JAN'
   WHEN InMonth = '02' THEN CharMonth = 'FEB'
   WHEN InMonth = '03' THEN CharMonth = 'MAR'
   WHEN InMonth = '04' THEN CharMonth = 'APR'
   WHEN InMonth = '05' THEN CharMonth = 'MAY'
   WHEN InMonth = '06' THEN CharMonth = 'JUN'
   WHEN InMonth = '07' THEN CharMonth = 'JUL'
   WHEN InMonth = '08' THEN CharMonth = 'AUG'
   WHEN InMonth = '09' THEN CharMonth = 'SEP'
   WHEN InMonth = '10' THEN CharMonth = 'OCT'
   WHEN InMonth = '11' THEN CharMonth = 'NOV'
   WHEN InMonth = '12' THEN CharMonth = 'DEC'
OTHERWISE Errortype = 5
END

IF ErrorType = 0
THEN
   DO
   RC = 0
   ChangeDate = Y
   ADDRESS COMMAND ,
'Date '||InDay||'-'||CharMonth||'-'||InYear
   IF RC = 0
   THEN
      NOP
   ELSE
      ErrorType = 6
   END
ELSE
```

```
   NOP

IF ErrorType = 0
THEN
   DO
   CALL DATE('Weekday')
   IF Result = 'Saturday'
   THEN
      ErrorType = 7
   ELSE
      IF Result = 'Sunday'
            THEN
               ErrorType = 8
            ELSE
               DO
                     InYearPrev = InYear
                     InMonthPrev = InMonth
                     InDayPrev = InDay
                     END
   END
ELSE
   NOP

RETURN


EDITLINE:
/* Edit line information                                    */

ErrorType = 0
StockName = FirstVar

IF Close = ' '
   THEN
   ErrorType = 20
   ELSE
   DO

   CALL DATATYPE(StockName,'M')
   IF Result = 1
      THEN
      NOP
   ELSE
      ErrorType = 9


   CALL DATATYPE(Volume)
   IF Result = NUM
      THEN
      NOP
   ELSE
      ErrorType = 12


   TempVar = High
   TempError = N
   CALL EDITPRICE
   IF TempError = Y
      THEN
      ErrorType = 13
      ELSE
      High = TempVar


   TempVar = Low
   TempError = N
   CALL EDITPRICE
   IF TempError = Y
      THEN
      ErrorType = 14
      ELSE
   Low = TempVar


   TempVar = Close
   TempError = N
   CALL EDITPRICE
   IF TempError = Y
      THEN
      ErrorType = 15
   ELSE
      Close = TempVar


   IF High < Low
      THEN
      ErrorType = 16
   ELSE
      IF Close < Low
         THEN
         ErrorType = 17
      ELSE
         IF Close > High
            THEN
                        ErrorType = 18
               ELSE
            NOP
   END

CALL EDITWARN

RETURN


EDITPRICE:
/* Edit stock prices                                        */

CALL DATATYPE(TempVar)
IF Result = NUM
   THEN
   PriceType = Integer
ELSE
   DO
   PriceType = NonInteger
   TVlength = (LENGTH(TempVar))
   TVlength = TVlength - 2
   TV1 = (LEFT(TempVar,TVlength))
   CALL DATATYPE(TV1)
   IF Result = NUM
      THEN
            DO
            Numerator = (RIGHT(TV1,1))
            Slash = (RIGHT(TempVar,2))
            Slash = (LEFT(Slash,1))
            Divisor = (RIGHT(TempVar,1))
            IF Slash = '/'
               THEN
                     CALL EDITFRACT
                     ELSE
                     TempError = Y
            END
   ELSE
      TempError = Y
   END

IF TempError = N
   THEN
   IF PriceType = Integer
      THEN
      NOP
   ELSE
      DO
            TVlength = TVlength - 1
            TV1 = (LEFT(TempVar,TVlength))
            TV2 = Numerator / Divisor
            TV2L = (LENGTH(TV2))
            TV2L = TV2L - 1
            TV2 = (RIGHT(TV2,TV2L))
            TempVar = TV1||TV2
            END


RETURN

EDITFRACT:
/* Make sure fraction is valid.                             */

IF Divisor = 2
   THEN
   IF Numerator = 1
      THEN
      NOP
   ELSE
      TempError = Y
ELSE
   IF Divisor = 4
      THEN
            DO
            SELECT
               WHEN Numerator = 1 THEN NOP
                     WHEN Numerator = 3 THEN NOP
                     OTHERWISE TempError = Y
          END
          END
          ELSE
             IF Divisor = 8
                      THEN
                      DO
          SELECT
              WHEN Numerator = 1 THEN NOP
                     WHEN Numerator = 3 THEN NOP
                              WHEN Numerator = 5 THEN NOP
                     WHEN Numerator = 7 THEN NOP
                     OTHERWISE TempError = Y
          END
          END
                    ELSE
                       TempError = Y

RETURN


EDITWARN:
/* Issue any warning messages.                              */
```

```
IF High > Low
   THEN
   DO
   Temp1 = High - Low
   Temp2 = Temp1 / High
   IF Temp2 > .15
      THEN
            DO
            ErrorType = 10
      OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
     CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
      END
   ELSE
      NOP
   END
ELSE
   NOP

RETURN


WRITELINE:
/* Put together line to go to sort.                          */

SymbolLength = (LENGTH(FirstVar))
FirstVar = (UPPER(FirstVar))

IF SymbolLength = 1
   THEN
   TempSymbol = FirstVar||'     '
ELSE
   IF SymbolLength = 2
      THEN
      TempSymbol = FirstVar||'    '
   ELSE
      IF SymbolLength = 3
         THEN
         TempSymbol = FirstVar||'   '
            ELSE
               IF SymbolLength = 4
         THEN
            TempSymbol = FirstVar||'  '
                  ELSE
            TempSymbol = FirstVar||' '


/* For some reason a padding blank is not needed between  */
/* "Low" & "Close". "Low" probably should have been       */
/* stripped of leading and trailing blanks somewhere      */
/* along the line.                                        */

OutString = ,
TempSymbol||InYear||InMonth||InDay||' '||InDay||'-',
||CharMonth||'-'||InYear||' '||Volume||' '||High||' ',
||Low||Close
CALL WRITELN('SortIn',OutString)

RETURN



TOTALPASS:
/* Get totals & dates for each stock and do final edit    */
/* for duplicates.                                        */

PARSE VAR InString Symbol SortDate Date Volume High Low ,
Close

IF PrevSymbol = Symbol
   THEN
   DO
   StockEntryCount = StockEntryCount + 1
   IF PrevSortDate = SortDate
      THEN
            DO
            ErrorType = 19
      OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
     CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
            ErrorFlag = Y
      END
   ELSE
      DO
      PrevSortDate = SortDate
            HoldLastDate = Date
            END
   END
ELSE
   IF StoxPresent = Y
      THEN
      DO
            OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
            CALL WRITELN('MsgFile',OutString)
      CALL WRITELN(StdOut,OutString)
            HoldFirstDate = Date
      HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END
   ELSE
      DO
            HoldFirstDate = Date
      HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END

StoxPresent = Y
InString = READLN('SortOut')
LineCount = LineCount + 1

RETURN
```

Listing Four

```
/*********************************************************/
/*                    ErrorReport                        */
/* This function provides standardized error processing  */
/* for StoxEdit and StoxEditGEnie. The input arguments   */
/* are a numeric error type, the line number of the input */
/* file in which the calling program has found the error, */
/* the input line in error, and the stock symbol related  */
/* to the input line in error. It formats an error        */
/* message, writes the message and the input line in      */
/* error to the console, and returns the error message to */
/* the calling program.                                   */
/*                                        Jack Fox        */
/*********************************************************/


ARG ErrorType,LineCount,InString,Symbol

/* Each error number has an associated message.          */
SELECT
   WHEN ErrorType = 1  THEN ErrorMsg = ,
'date not numeric'
   WHEN ErrorType = 2  THEN ErrorMsg = ,
'date less than previous date'
   WHEN ErrorType = 3  THEN ErrorMsg = ,
'date same as previous date'
   WHEN ErrorType = 4  THEN ErrorMsg = ,
'date greater than current date'
   WHEN ErrorType = 5  THEN ErrorMsg = ,
'date month invalid'
   WHEN ErrorType = 6  THEN ErrorMsg = ,
'date invalid'
   WHEN ErrorType = 7  THEN ErrorMsg = ,
'date is a Saturday'
   WHEN ErrorType = 8  THEN ErrorMsg = ,
'date is a Sunday'
   WHEN ErrorType = 9  THEN ErrorMsg = ,
'stock name is invalid'
   WHEN ErrorType = 10 THEN ErrorMsg = ,
'WARNING: difference between High & Low GT 15%'
   WHEN ErrorType = 12 THEN ErrorMsg = ,
'volume not numeric'
   WHEN ErrorType = 13 THEN ErrorMsg = ,
'high price invalid'
   WHEN ErrorType = 14 THEN ErrorMsg = ,
'low price invalid'
   WHEN ErrorType = 15 THEN ErrorMsg = ,
'closing price invalid'
   WHEN ErrorType = 16 THEN ErrorMsg = ,
'High is less than Low'
   WHEN ErrorType = 17 THEN ErrorMsg = ,
'Close is less than Low'
   WHEN ErrorType = 18 THEN ErrorMsg = ,
'Close is greater than High'
   WHEN ErrorType = 19 THEN ErrorMsg = ,
'Duplicate date for stock '||Symbol||' in RAM:Stox'
   WHEN ErrorType = 20 THEN ErrorMsg = ,
'Not enough arguments'
   OTHERWISE ErrorMsg = 'unspecified error'
END

/* Build the error message including line number.        */
OutString = '==> Line '||LineCount||' '||ErrorMsg
CALL WRITELN(StdOut,OutString)
CALL WRITELN(StdOut,InString)

/* Return error message to calling program.              */
EXIT OutString
```

Listing Five

```
/*************************************************************/
/*                          TotalPass                        */
/*                                                           */
/* The output file, which is input to program StoxUpdate     */
/* is sorted by stock symbol and date so that all updates    */
/* to a given spreadsheet take place together while that     */
/* plan is only opened once. An output line looks like       */
/* this:                                                     */
/*                                                           */
/* ACN  900629 29-JUN-90 357 39.25 38.75 38.75               */
/*                                                           */
/* Error messages and counts of activity lines for each      */
/* individual stock are written both to a message file       */
/* and to the console.                                       */
/*                                      Jack Fox             */
/*************************************************************/

/* Get file name parameter entered on the command line and*/
/* make sure that file exists.                               */
Arg Filename

IF Filename = ''
   THEN DO
   SAY 'Please enter input file name as parameter.'
   CALL EOJ
   END
ELSE
   IF EXISTS(Filename)
   THEN
      SAY 'using file '||Filename
   ELSE
      DO
      Filename = 'T:'||Filename
            IF EXISTS(Filename)
            THEN
               SAY 'using file '||filename
            ELSE
               DO
               Say Filename||' does not exist'
                     CALL EOJ
                     END
      END

/* Open the message file.                                    */
CALL OPEN('MsgFile','T:StoxEdit.msg','write')

/* Sort the input file then open the sorted file as input.*/
ADDRESS COMMAND 'SORT 'Filename' T:Stox'
CALL OPEN('SortOut','T:Stox','read')

/* Read first line of sorted data.                           */
InString=READLN('SortOut')
LineCount = 1
StockEntryCount = 1
PrevSymbol = ' '


DO WHILE ~ EOF('SortOut')
   CALL TOTALPASS
END

/* If there was any data, write message line for last        */
/* stock processed.                                          */
IF StoxPresent = Y
   THEN
   DO
   OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
  CALL WRITELN('MsgFile',OutString)
  CALL WRITELN(StdOut,OutString)
   END
ELSE
   NOP


EOJ:
/* End of the program.                                       */

CALL CLOSE('MsgFile')
SAY 'Job Ended'
EXIT

TOTALPASS:
/* Get totals & dates for each stock and do final edit       */
/* for duplicates.                                           */

PARSE VAR InString Symbol SortDate Date EPSrnk RelStr ,
Volume High Low Close

IF PrevSymbol = Symbol
   THEN
   DO
   StockEntryCount = StockEntryCount + 1
   IF PrevSortDate = SortDate
      THEN
            DO
            ErrorType = 19
            OutString = ,
(ErrorReport(ErrorType,LineCount,InString,Symbol))
            CALL WRITELN('MsgFile',OutString)
     CALL WRITELN('MsgFile',InString)
            END
   ELSE
      DO
      PrevSortDate = SortDate
            HoldLastDate = Date
            END
   END
ELSE
   IF StoxPresent = Y
      THEN
      DO
            OutString = ,
PrevSymbol||' '||StockEntryCount||' entries from ',
||HoldFirstDate||' to '||HoldLastDate
            CALL WRITELN('MsgFile',OutString)
     CALL WRITELN(StdOut,OutString)
            HoldFirstDate = Date
      HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END
   ELSE
      DO
            HoldFirstDate = Date
      HoldLastDate = Date
            PrevSortDate = SortDate
            PrevSymbol = Symbol
            StockEntryCount = 1
            END

StoxPresent = Y
InString = READLN('SortOut')
LineCount = LineCount + 1

RETURN
```

Listing Six

```
/*************************************************************/
/*                 StartAdv.rexx                             */
/*                                                           */
/* Initiates The Advantage spreadsheet and loads all of      */
/* the macros used by the spreadsheet in other ARexx         */
/* programs.                                                 */
/*                                      Jack Fox             */
/*************************************************************/

/* Start the spreadsheet program.                            */
Pgm = 'DH0:Advantage/Advantage'
ADDRESS COMMAND 'Run' Pgm '-i'

/* Wait 5 seconds (using Scripit) while the program loads */
/* before trying to communicate with it.                     */
ADDRESS XIT
'WAIT 250'

/* Let ARexx know to send unrecognized lines to The       */
/* Advantage.                                                */
ADDRESS 'Advantage'
OPTIONS results

/* Load The Advantage macros into the program.            */

Macro = 'DH0:Advantage/Macros/calc'
'LoadMacros'
   Macro

Macro = 'DH0:Advantage/Macros/fill_down'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/formula_to_value'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/freeze_title'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/paste_row'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/select_all'
'LoadMacros'
  Macro
Macro = 'DH0:Advantage/Macros/copy'
```

```
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/paste'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/delete_column'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/chart_1'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/chart_2'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/chart_3'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/chart_4'
'LoadMacros'
  Macro

Macro = 'DH0:Advantage/Macros/chart_5'
'LoadMacros'
  Macro

EXIT
```

Listing Seven

```
/*********************************************************/
/*                      StoxUpdate                       */
/*                                                       */
/* This program takes the output from StoxEdit and adds  */
/* the data for each stock to its own spreadsheet. If a  */
/* stock spreadsheet does not exist, it is created from  */
/* the model spreadsheet.                                */
/*                                                       */
/* Spreadsheets are stored in ZOOed format to save disk  */
/* space. ZOO is used because it arcs faster than  any of */
/* the more modern, more efficient arc programs.         */
/*                                                       */
/* The only argument to this program is "Device", which  */
/* is the device or path that the spreadsheets and model */
/* plan reside on. If not entered it defaults to "Stox:". */
/*                                                       */
/*                                    Jack Fox           */
/*********************************************************/

ARG Device
LineCount = 1
Device = 'STOX:'
TempDir = 'Temp:'
ModelPlan = Device||'Model.adv'

CALL OPEN('LogFile',Device||'adv.log','write')
CALL OPEN('WrnFile',Device||'adv.wrn','write')

ADDRESS 'Advantage'
OPTIONS results

/* Make sure input file exists.                          */
IF ~ EXISTS('T:Stox')
   THEN
   DO
   MsgLine = 'T:Stox does not exist - task ended'
   SAY MsgLine
   CALL WRITELN('LogFile',MsgLine)
   CALL EOJ
   END
ELSE
   NOP

/* Make sure script file exists.                         */
IF ~ EXISTS('S:StoxScript')
   THEN
   DO
   MsgLine = 'S:StoxScript does not exist - task ended'
   SAY MsgLine
   CALL WRITELN('LogFile',MsgLine)
   CALL EOJ
   END
ELSE
   NOP

/* Read the first input record and parse it.             */
CALL OPEN('InFile','T:Stox','read')
InString = READLN('InFile')
PARSE VAR InString HoldSymbol RestOfLine

/* Establish "T:" as the current directory.              */
CALL PRAGMA('Directory','T:')

/* Formulas for building new spreadsheets.               */
E23 = '=(SUM(D4:D23)/20)'
F53 = '=(SUM(D4:D53)/50)'
H5 = '=IF(D5>D4,G5,0)'
I5 = '=IF(D5<D4,G5,0)'
J9 = '=IF((IF(SUM(H5:H9)>0,SUM(H5:H9),5)/IF(SUM(I5:I9)>0,
SUM(I5:I9),5))>19.99,19.99,(IF(SUM(H5:H9)>0,SUM(H5:H9),5)
/IF(SUM(I5:I9)>0,SUM(I5:I9),5)))'
K24 = '=IF((IF(SUM(H5:H24)>0,SUM(H5:H24),20)/IF(SUM(I5:I24)
>0,SUM(I5:I24),20))>4.99,4.99,(IF(SUM(H5:H24)>0,
SUM(H5:H24),20)/IF(SUM(I5:I24)>0,SUM(I5:I24),20)))'
L54 = '=(IF(SUM(H5:H54)>0,SUM(H5:H54),50)/IF(SUM(I5:I54)>0,
SUM(I5:I54),50))'
M53 = '=(SUM(G4:G53)/50)'
N53 = '=(IF((((G53-M53)/M53)*100)<499,(((G53-M53)/M53)*100)
,499))'


/* Get date and write it to log file.                    */
SaveDate = (DATE('normal'))
PARSE VAR SaveDate Day Month Year
SaveYear = (RIGHT(Year,2))

Weekday = (DATE('Weekday'))
SaveDay = (LEFT(Weekday,3))

LogDate = SaveDay||' '||Day||'-'||Month||'-'||SaveYear

CALL WRITELN('LogFile',LogDate)
CALL CLOSE('LogFile')
CALL CLOSE('WrnFile')
CALL OPEN('IndFile',Device||'adv.ind','write')
CALL CLOSE('IndFile')

/* Go do the main process until done with the input file. */
DO WHILE ~ EOF('InFile')
   CALL MAIN_PROCESS
END

EOJ:
/* Write the last LogFile line and quit.                 */
MsgLine = 'StoxUpdate ended'
SAY MsgLine
CALL OPEN('LogFile',Device||'adv.log','append')
CALL WRITELN('LogFile',MsgLine)
CALL CLOSE('LogFile')

EXIT


MAIN_PROCESS:
/* Process the data for one stock.                       */

/* Comment next line if using ConMan.                    */
/* If NOT using ConMan, we're going to use T:WorkFile to */
/* queue up the data for one stock.                      */
CALL OPEN('WorkFile','T:WorkFile','write')

CALL OPEN('LogFile',Device||'adv.log','append')
CALL OPEN('WrnFile',Device||'adv.wrn','append')

/* Queue up the data for one stock and count the number  */
/* of data lines for that stock.                         */
LineCount = 0
DO WHILE ~ EOF('InFile')
   CALL QUEUE_UP
   IF HoldSymbol ~= Symbol
      THEN
            LEAVE
   ELSE
      NOP
END

/* Comment next 2 lines if using ConMan.                 */
CALL CLOSE('WorkFile')
CALL OPEN('WorkFile','T:WorkFile','read')

PlanZoo = Device||HoldSymbol||'.zoo'
RamZoo = 'T:'||HoldSymbol||'.zoo'
PlanName = 'T:'||HoldSymbol||'.adv'
NewPlan = N

/* If a ZOOed spreadsheet dosen't already exist for this */
/* stock, we need to use a copy of the model spreadsheet. */
/* Otherwise we extract the spreadsheet from the ZOOed   */
/* file to the work area in T:.                          */
IF EXISTS(PlanZoo)
   THEN
   DO
   ADDRESS COMMAND 'T:Zoo e' PlanZoo
   ADDRESS COMMAND 'Run Copy' PlanZoo TempDir
   END
ELSE
   DO
   NewPlan = Y
```

```
   IF MODEL_IN_RAM = Y
      THEN
     ADDRESS COMMAND 'COPY' ModelPlan PlanName
   ELSE
      IF ~ EXISTS(ModelPlan)
         THEN
         DO
        MsgLine = ModelPlan||' does not exist - task ended'
         SAY MsgLine
        CALL WRITELN('LogFile',MsgLine)
         CALL EOJ
         END
      ELSE
         DO
        ADDRESS COMMAND 'COPY' ModelPlan 'T:'
              ModelPlan = 'T:Model.adv'
        ADDRESS COMMAND 'COPY' ModelPlan PlanName
        MODEL_IN_RAM = Y
              END
   END

/* Go process the spreadsheet.                              */
CALL PROCESS_PLAN

/* Comment next line if using ConMan.                       */
CALL CLOSE('WorkFile')

IF ~EOF('InFile')
   THEN
   HoldSymbol = NextSymbol
ELSE
   NOP

/* ZOO the completed spreadsheet, clean up the work area    */
/* and copy the ZOOed file back to STOX:.                   */
ADDRESS COMMAND 'T:Zoo a'  RamZoo PlanName
ADDRESS COMMAND 'Run Delete' PlanName
ADDRESS COMMAND 'Run Delete' PlanName||'.info'
/* StoxScript copies the ZOOed file and deletes it.         */
ADDRESS COMMAND 'Run Execute S:StoxScript' RamZoo Device

/* Write a line to the LogFile saying we processed data     */
/* for this stock and which days we processed.              */
LogLine = LogSymbol||' '||FirstDay||' - '||DDMMMYY
CALL WRITELN('LogFile',LogLine)
CALL CLOSE('LogFile')
CALL CLOSE('WrnFile')

RETURN


QUEUE_UP:
/* Queues all the lines for a single stock.                 */

/* Uncomment next line if using ConMan.                     */
/*QUEUE InString*/
/* Comment next line if using ConMan.                       */
CALL WRITELN('WorkFile',InString)

InString = READLN('InFile')
PARSE VAR InString Symbol RestOfLine

IF HoldSymbol = Symbol
   THEN
   LineCount = LineCount + 1
ELSE
   NextSymbol = Symbol

RETURN


PROCESS_PLAN:
/* Opens, processes & closes a spreadsheet.                 */

/* Open the spreadsheet in The Advantage.                   */
OpenSymbol = 'T:'||HoldSymbol||'.adv'
'LoadSpread'
  OpenSymbol

/* Uncomment next line if using ConMan.                     */
/*PULL PullString*/
/* Comment next line if using ConMan.                       */
PullString = READLN('WorkFile')

Count = LineCount
PARSE VAR PullString Symbol JulianDate DDMMMYY Vol High ,
Low Close .

/* Get rid of spaces around date.                          */
CALL TRIM(JulianDate)
CALL DATATYPE(JulianDate)
IF Result = NUM
   THEN
   NOP
ELSE
   DO
   MsgLine = 'Fatal error in input data for '||Symbol
   SAY MsgLine
   CALL WRITELN('LogFile',MsgLine)
   CALL EOJ
   END

CALL TRIM(Close)

/* Format the date for use by "all time high" & "low".      */
PARSE VAR DDMMMYY DD '-' MMM '-' YY
MMMDDYY = MMM||'-'||DD||'-'||YY

NewAllTimeHigh = N
NewAllTimeLow = N

FirstDay = DDMMMYY
LogSymbol = Symbol

/* If this is a new spreadsheet, set the "all time high"    */
/* & "low" so that we determine a high & low. Then do the   */
/* routines for new spreadsheets.                           */
IF NEWPLAN = Y
   THEN
   DO
   AllTimeHigh = 0
   AllTimeLow = 9999
   CALL  NEW_PLAN_INIT
   CALL PLACE_DATA
      DO WHILE Count ~= 0
         CALL PULL_ROUTINE
      END
   CALL NEW_PLAN_FORMULAS
   END
ELSE
/* If this is for an existing spreadsheet, we will extract  */
/* the high & low from the spreadsheet and then do the      */
/* routine for existing spreadsheets.                       */
   DO
   CellName = 'O5'
   'SelectCell'
      CellName
   'GetValue'
   AllTimeHigh = Result
   CellName = 'O7'
   'SelectCell'
      CellName
   'GetValue'
   AllTimeLow = Result
   CALL PROCESS_OLD_PLAN
   END

/* If we found an "all time high", then update it.          */
IF NewAllTimeHigh = Y
THEN
   DO
   CellName = 'O5'
   'SelectCell'
      CellName
   'PutCell'
      AllTimeHigh
   CellName = 'O4'
   'SelectCell'
      CellName
   'PutCell'
      AllTimeHighDate
   END
ELSE
   NOP

/* If we found an "all time low", then update it.           */
IF NewAllTimeLow = Y
THEN
   DO
   CellName = 'O7'
   'SelectCell'
      CellName
   'PutCell'
      AllTimeLow
   CellName = 'O6'
   'SelectCell'
      CellName
   'PutCell'
      AllTimeLowDate
   END
ELSE
   NOP

/* Open the indicator file, then test whether we should     */
/* report anything should be reported about this stock.     */
CALL OPEN('IndFile',Device||'adv.ind','append')

CALL TEST_INDICATORS

CALL CLOSE('IndFile')

/* Save the spreadsheet.                                    */
'SaveSpread'
  OpenSymbol

RETURN
```

```
NEW_PLAN_INIT:
/* Initialize a new spreadsheet.                          */

/* Put the stock symbol in cell A1.                       */
CellName = 'A1'
'SelectCell'
   CellName
'PutCell'
   Symbol

/* Freeze the first row.                                  */
macro = 'freeze_title'
'macro'
  macro

/* We will start entering data in row 4.                  */
RowNumber = 4

RETURN


PULL_ROUTINE:
/* Pull a line from the queue and parse it.               */

/* Uncomment next line if using ConMan.                   */
/*PULL PullString*/
/* Comment next line if using ConMan.                     */
PullString = READLN('WorkFile')

/* As we pull a line, decrement the count.                */
Count = Count - 1
PARSE VAR PullString Symbol JulianDate DDMMMYY Vol High ,
Low Close .
/* Dates come in old MaxiPlan format, switch to Advantage.*/
CALL TRIM(Close)
PARSE VAR DDMMMYY DD '-' MMM '-' YY
MMMDDYY = MMM||'-'||DD||'-'||YY

/* Point to the next spreadsheet row...                   */
RowNumber = RowNumber + 1

/* ...and put the data in it.                             */
CALL PLACE_DATA

RETURN


PLACE_DATA:
/* Inserts data in a row, edits Close against previous    */
/* values and issues warning message if difference is     */
/* suspicious, and updates AllTimeHigh & Low.             */

CellName = 'A'||RowNumber
'SelectCell'
   CellName

/* The date in column A is going to be a 4 digit date,    */
/* so that when data is graphed the dates labeling the    */
/* horizontal ticks overlaps as little as possible.       */
MMDD = (RIGHT(JulianDate,4))

PasteRow = '"'||MMDD||'","'||High||'","'||Low||'",'
||Close||'"," "," ","'||Vol||'"," "," "," "," "," "," "'

/* Write a line of data to a temporary file so that it    */
/* can be pasted to the spreadsheet.                      */
CALL OPEN('PasteFile','T:PasteFile','write')
CALL WRITELN('PasteFile',PasteRow)
CALL CLOSE('PasteFile')

macro = 'paste_row'
'macro'
  macro

/* Test whether we have a new high at the close.          */
IF Close > AllTimeHigh
THEN
   DO
   NewAllTimeHigh = Y
   AllTimeHigh = Close
   AllTimeHighDate = MMMDDYY
   END
ELSE
   NOP

/* Test whether we have a new low at the close.           */
IF Close < AllTimeLow
THEN
   DO
   NewAllTimeLow = Y
   AllTimeLow = Close
   AllTimeLowDate = MMMDDYY
   END
ELSE
   NOP

/* Test whether new data is "out of line" with old data.  */
IF RowNumber > 6
   THEN
   CALL TESTDATA
ELSE
   NOP


RETURN


NEW_PLAN_FORMULAS:
/* Put formulas in a new plan, calculate, then convert    */
/* formulas to data except the last row to save space.    */

CellName = 'E23'
'SelectCell'
   CellName
'PutCell'
   E23

CellName = 'F53'
'SelectCell'
   CellName
'PutCell'
   F53

CellName = 'H5'
'SelectCell'
   CellName
'PutCell'
   H5

CellName = 'I5'
'SelectCell'
   CellName
'PutCell'
   I5

CellName = 'J9'
'SelectCell'
   CellName
'PutCell'
   J9

CellName = 'K24'
'SelectCell'
   CellName
'PutCell'
   K24

CellName = 'L54'
'SelectCell'
   CellName
'PutCell'
   L54

CellName = 'M53'
'SelectCell'
   CellName
'PutCell'
   M53

CellName = 'N53'
'SelectCell'
   CellName
'PutCell'
   N53

TopCorner = 'E23'
BottomCorner ='E'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'F53'
BottomCorner ='F'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'H5'
BottomCorner ='H'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro
```

```
TopCorner = 'I5'
BottomCorner ='I'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'J9'
BottomCorner ='J'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'K24'
BottomCorner ='K'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'L54'
BottomCorner ='L'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'M53'
BottomCorner ='M'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'N53'
BottomCorner ='N'||RowNumber
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calc'
'macro'
  macro

CellName = 'E23'
'SelectCell'
   CellName
'GetValue'
Value = result
CellName = 'E4'
'SelectCell'
   CellName
'PutCell'
   Value

CellName = 'F53'
'SelectCell'
   CellName
'GetValue'
Value = result
CellName = 'F4'
'SelectCell'
   CellName
'PutCell'
   Value

CellName = 'H5'
'SelectCell'
   CellName
macro = 'copy'
'macro'
  macro
CellName = 'H4'
'SelectCell'
   CellName

CellName = 'I5'
'SelectCell'
   CellName
macro = 'copy'
'macro'
  macro
CellName = 'I4'
'SelectCell'
   CellName

CellName = 'J4'
'SelectCell'
   CellName
'PutCell'
   '1'

CellName = 'K4'
'SelectCell'
   CellName
'PutCell'
   '1'

CellName = 'L4'
'SelectCell'
   CellName
'PutCell'
   '1'

TopCorner = 'E4'
BottomCorner ='E22'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'F4'
BottomCorner ='F52'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'J4'
BottomCorner ='J8'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'K4'
BottomCorner ='K23'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'L4'
BottomCorner ='L53'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

CellName = 'N4'
CellValue = 0
'SelectCell'
   CellName
'PutCell'
   CellValue

TopCorner = 'N4'
BottomCorner ='N52'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

RowNumber = RowNumber - 1
RowStart = 4

IF RowNumber > 103
   THEN
   RowEnd = 103
ELSE
   RowEnd = RowNumber

DO WHILE RowStart ~> RowNumber
   CALL PASTE_DATA
END

RETURN


PROCESS_OLD_PLAN:
```

```
/* Process an existing spreadsheet.                          */

/* Determine what the row number of the last row is.         */
macro = 'select_all'
'macro'
  macro
'current'
  temp = result

PARSE VAR temp wk1 ':' wk2

RowNumber = (SUBSTR(wk2,2))

RowStart = RowNumber
RowEnd = RowStart + Count + 1
RowNumber = RowNumber + 1

CALL PLACE_DATA

/* Make sure the first date added is greater than the        */
/* last date in the spreadsheet.                             */
CompareRow = RowNumber - 1

CellName = 'A'||CompareRow
'SelectCell'
   CellName

'GetValue'
CBData = result

GoodDate = Y

IF MMDD ~> CBData
THEN
   DO
   TempMM = (LEFT(MMDD,2))
   TempDD = (RIGHT(MMDD,2))
   TempTen = (LEFT(TempDD,1))
   TempDay = (RIGHT(TempDD,1))
   IF TempMM = '01'
   THEN
      DO
            IF TempTen = 0
            THEN
               DO
                     IF TempDay < 5
                     THEN
                        NOP
                     ELSE
                        GoodDate = N
                     END
            ELSE
               GoodDate = N
            END
   ELSE
      GoodDate = N
   END
ELSE
   NOP

IF GoodDate = N
THEN
   DO
   ErrorMsg = 'FATAL ERROR: first date added not greater
than last date'
   DataMsg = Symbol||'  '||MMDD||'  previous data: '||CBData
   CALL WRITELN('WrnFile',ErrorMsg)
   CALL WRITELN(StdOut,ErrorMsg)
   CALL WRITELN('WrnFile',DataMsg)
   CALL WRITELN(StdOut,DataMsg)
    CALL EOJ
    END
ELSE
   NOP

/* Enter the new data.                                       */
DO WHILE Count ~= 0
   CALL PULL_ROUTINE
END

/* Fill down the equations to the rows containing new        */
/* data.                                                     */
TopCorner = 'E'||RowStart
BottomCorner ='F'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'H'||RowStart
BottomCorner ='L'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'M'||RowStart
BottomCorner ='N'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

/* Recalculate the new formulas.                             */
macro = 'calc'
'macro'
  macro

/* Change all the formulas to values, except the formulas */
/* in the last row, which will be propogated the next time*/
/* we add data.                                              */
RowNumber = RowNumber - 1
RowEnd = RowNumber
Wk4 = RowEnd - RowStart
IF Wk4 > 99
   THEN
   RowEnd = RowStart + 99
ELSE
   NOP

DO WHILE RowStart ~> RowNumber
   CALL PASTE_DATA
END

RETURN


PASTE_DATA:
/* Replace formulas with calculated values for all but       */
/* last row.                                                 */

TopCorner = 'E'||RowStart
BottomCorner ='N'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner

macro = 'formula_to_value'
'macro'
  macro

RowStart = RowStart + 100
RowEnd = RowEnd + 100

IF RowEnd > RowNumber
   THEN
   RowEnd = RowNumber
ELSE
   NOP

RETURN


TESTDATA:
/* See if the diverges by more than 15% and issue a          */
/* warning message if it does.                               */

CompareRow = RowNumber - 1

CellName = 'D'||CompareRow
'SelectCell'
   CellName

'GetValue'
CBData = result

IF CBData > Close
   THEN
   DO
   High = CBData
   Low = Close
   END
ELSE
   DO
   High = Close
   Low = CBData
   END

Temp1 = High - Low
Temp2 = Temp1 / High
IF Temp2 > .15
   THEN
   DO
   IF Temp1 > 4
   THEN
      DO
      ErrorMsg = 'WARNING: Close GT 15% difference ',
||DDMMMYY
```

```
      DataMsg = Symbol||'  '||Close||'  previous data: ',
||CBData
            BlankLine = ' '
      CALL WRITELN('WrnFile',ErrorMsg)
      CALL WRITELN(StdOut,ErrorMsg)
      CALL WRITELN('WrnFile',DataMsg)
      CALL WRITELN(StdOut,DataMsg)
      CALL WRITELN('WrnFile',BlankLine)
      CALL WRITELN(StdOut,BlankLine)
       END
   ELSE
      NOP
   END
ELSE
   NOP

RETURN


TEST_INDICATORS:
/* Report whether the stock closed within 15% of its      */
/* "all time high" or whether it is exhibiting a surge in */
/* upside volume.                                         */

CellName = 'K'||RowNumber
'SelectCell'
   CellName

'GetValue'
TwentyDayUpDown = result

IF TwentyDayUpDown = 4.99
THEN
   DO
   IndMsg = SYMBOL||' 20 Day Up/Down is '||TwentyDayUpDown
   CALL WRITELN('IndFile',IndMsg)
   END
ELSE
   NOP

Temp1 = AllTimeHigh - Close
Temp2 = Temp1 / AllTimeHigh
IF Temp2 < .151
   THEN
   DO
   Temp3 = (LEFT(Temp2,5))
   Temp4 = (RIGHT(Temp3,3))
   Temp5 = (LEFT(Temp4,2))
   Temp6 = (LEFT(Temp5,1))
   Temp7 = (RIGHT(Temp5,1))
   Temp8 = (Right(Temp4,1))
   IF Temp6 = 0
      THEN
            Percent = Temp7||'.'||Temp8
   ELSE
      Percent = Temp5||'.'||Temp8
   IndMsg = SYMBOL||' is '||Percent||' % from All Time High'
   CALL WRITELN('IndFile',IndMsg)
   END
ELSE
   NOP

RETURN
```

Listing Eight

```
.KEY RamZoo,Device
COPY <RamZoo> <Device>
DELETE <RamZoo>
```

Listing Nine

```
/*********************************************************/
/*                    StoxSplit.rexx                     */
/*                                                       */
/* This program readjusts an existing spreadsheet for    */
/* stock splits. Arguments to this program are:          */
/*                                                       */
/*     stock symbol                                      */
/*     type of split (2/1, 3/1, 4/1, or 3/2)             */
/*     row number the split occurred on                  */
/*                                                       */
/* The high, low, and close are adjusted for all the rows */
/* PREVIOUS to the row on which the split occurred. Then  */
/* the moving averages of the closing price is adjusted.  */
/*                                        Jack Fox       */
/*********************************************************/


ARG Symbol Split SplitRow
SplitRow = STRIP(SplitRow)

ADDRESS 'Advantage'

OPTIONS results

CALL PRAGMA('Directory','T:')

Device = 'STOX:'
TempZoo = 'TEMP:'
PlanZoo = 'STOX:'||Symbol||'.zoo'
RamZoo = 'T:'||Symbol||'.zoo'
PlanName = 'T:'||Symbol||'.adv'

/* Define formulas.                                       */
E23 = '=(SUM(D4:D23)/20)'
F53 = '=(SUM(D4:D53)/50)'
U2for1 = '=(B4/2)'
V2for1 = '=(C4/2)'
W2for1 = '=(D4/2)'
U3for1 = '=(B4/3)'
V3for1 = '=(C4/3)'
W3for1 = '=(D4/3)'
U4for1 = '=(B4/4)'
V4for1 = '=(C4/4)'
W4for1 = '=(D4/4)'
U3for2 = '=((B4*2)/3)'
V3for2 = '=((C4*2)/3)'
W3for2 = '=((D4*2)/3)'

CALL OPEN('MsgFile','T:StoxSplit.msg','write')

IF Split = '2/1'
   THEN
   NOP
ELSE
   IF Split = '3/2'
      THEN
      NOP
   ELSE
      IF Split = '3/1'
         THEN
         NOP
      ELSE
         IF Split = '4/1'
            THEN
            NOP
         ELSE
            DO
            MsgLine = 'split invalid  - task ended'
            SAY MsgLine
            CALL WRITELN('MsgFile',MsgLine)
            CALL EOJ
            END

IF EXISTS(Planzoo)
   THEN
   DO
   ADDRESS COMMAND 'COPY' PlanZoo TempZoo
   ADDRESS COMMAND 'COPY C:zoo T:'
   ADDRESS COMMAND 'T:zoo e' PlanZoo
   END
ELSE
   DO
   MsgLine = 'zooed plan does not exist  - task ended'
   SAY MsgLine
   CALL WRITELN('MsgFile',MsgLine)
   CALL EOJ
   END

'LoadSpread'
  PlanName

/* Finde the last row number.                             */
macro = 'select_all'
'macro'
  macro
'current'
  temp = result

PARSE VAR temp wk1 ':' wk2

RowNumber = (SUBSTR(wk2,2))

IF SplitRow ~< RowNumber
   THEN
   DO
   MsgLine = 'SplitRow invalid  - task ended'
   SAY MsgLine
   CALL WRITELN('MsgFile',MsgLine)
   CALL EOJ
   END
ELSE
   IF SplitRow ~> 4
      THEN
      DO
            MsgLine = 'SplitRow invalid  - task ended'
      SAY MsgLine
      CALL WRITELN('MsgFile',MsgLine)
      CALL EOJ
      END
   ELSE
```

```
      NOP

CALL MAIN_PROCESS


EOJ:
/* End of program.                                          */

MsgLine = 'StoxSplit ended'
SAY MsgLine
CALL WRITELN('MsgFile',MsgLine)

EXIT



MAIN_PROCESS:
/*                                                          */

LastSplit = SplitRow - 1

IF Split = '2/1'
   THEN
   CALL BUILD_2FOR1
ELSE
   IF Split = '3/1'
      THEN
      CALL BUILD_3FOR1
   ELSE
      IF Split = '4/1'
         THEN
         CALL BUILD_4FOR1
      ELSE
         CALL BUILD_3FOR2

RowStart = 4

IF LastSplit < 104
   THEN
   RowEnd = LastSplit
ELSE
   RowEnd = 103

DO WHILE RowStart ~>  LastSplit
    CALL PASTE_PRICE_DATA
END

/* Delete the work columns.                                 */
TopCorner = 'U4'
BottomCorner = 'W4'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'delete_column'
'macro'
  macro

/* Recreate the moving close averages.                      */
CellName = 'E23'
'SelectCell'
   CellName
'PutCell'
   E23

CellName = 'F53'
'SelectCell'
   CellName
'PutCell'
   F53

RowNumber = RowNumber - 1

/* Determine how many rows we need to recreate the          */
/* averages for.                                            */
LastE = LastSplit + 20
IF LastE > RowNumber
   THEN
   LastE = RowNumber
ELSE
   NOP

LastF = LastSplit + 50
IF LastF > RowNumber
   THEN
   LastF = RowNumber
ELSE
   NOP

TopCorner = 'E23'
BottomCorner = 'E'||LastE
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'F53'
BottomCorner = 'F'||LastF
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calculate'
'macro'
  macro

CellName = 'E23'
'SelectCell'
   CellName
'GetValue'
Value = result
CellName = 'E4'
'SelectCell'
   CellName
'PutCell'
   Value

CellName = 'F53'
'SelectCell'
   CellName
'GetValue'
Value = result
CellName = 'F4'
'SelectCell'
   CellName
'PutCell'
   Value

TopCorner = 'E4'
BottomCorner ='E22'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

TopCorner = 'F4'
BottomCorner ='F52'
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

RowStart = 23

IF LastF < 123
   THEN
   RowEnd = LastF
ELSE
   RowEnd = 122

DO WHILE RowStart ~>  LastF
    CALL PASTE_AVG_DATA
END

RowNumber = RowNumber + 1
AllTimeHigh = 0
AllTimeLow = 999
CurrentRow = 4

DO WHILE CurrentRow ~> RowNumber
   CALL FIND_HIGH_LOW
END

CellName = 'O4'
'SelectCell'
   CellName
'PutCell'
   AllTimeHighDate

CellName = 'O5'
'SelectCell'
   CellName
'PutCell'
   AllTimeHigh

CellName = 'O6'
'SelectCell'
   CellName
'PutCell'
   AllTimeLowDate

CellName = 'O7'
'SelectCell'
   CellName
'PutCell'
   AllTimeLow

'SaveSpread'
```

```
  PlanName


/* ZOO the completed spreadsheet, clean up the work area  */
/* and copy the ZOOed file back to STOX:.                 */
ADDRESS COMMAND 'T:Zoo a'  RamZoo PlanName
ADDRESS COMMAND 'Run Delete' PlanName
ADDRESS COMMAND 'Run Delete' PlanName||'.info'
/* StoxScript copies the ZOOed file and deletes it.       */
ADDRESS COMMAND 'Run Execute S:StoxScript' RamZoo Device

RETURN


PASTE_PRICE_DATA:
/*                                                        */

TopCorner = 'U'||RowStart
BottomCorner ='W'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner

macro = 'formula_to_value'
'macro'
  macro

macro = 'copy'
'macro'
  macro

CellName = 'B'||RowStart
'SelectCell'
   CellName

macro = 'paste'
'macro'
  macro

RowStart = RowStart + 100
RowEnd = RowEnd + 100

IF RowEnd > LastSplit
   THEN
   RowEnd = LastSplit
ELSE
   NOP

RETURN



PASTE_AVG_DATA:
/* Change the averages formulas to values.               */

TopCorner = 'E'||RowStart
BottomCorner ='F'||RowEnd
'SelectRange'
  TopCorner
  BottomCorner

macro = 'formula_to_value'
'macro'
  macro

RowStart = RowStart + 100
RowEnd = RowEnd + 100

IF RowEnd > LastF
   THEN
   RowEnd = LastF
ELSE
   NOP

RETURN


BUILD_2FOR1:
/* Process 2 for 1 split.                                */

CellName = 'U4'
'SelectCell'
   CellName
'PutCell'
   U2for1

CellName = 'V4'
'SelectCell'
   CellName
'PutCell'
   V2for1

CellName = 'W4'
'SelectCell'
   CellName
'PutCell'
   W2for1

TopCorner = 'U4'
BottomCorner ='W'||LastSplit
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calculate'
'macro'
  macro

CellName = 'O'||SplitRow
'SelectCell'
   CellName
'PutCell'
   "split 2/1"

RETURN


BUILD_3FOR1:
/* Process 3 for 1 split.                                */

CellName = 'U4'
'SelectCell'
   CellName
'PutCell'
   U3for1

CellName = 'V4'
'SelectCell'
   CellName
'PutCell'
   V3for1

CellName = 'W4'
'SelectCell'
   CellName
'PutCell'
   W3for1

TopCorner = 'U4'
BottomCorner ='W'||LastSplit
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calculate'
'macro'
  macro

CellName = 'O'||SplitRow
'SelectCell'
   CellName
'PutCell'
   "split 3/1"

RETURN


BUILD_4FOR1:
/* Process 4 for 1 split.                                */

CellName = 'U4'
'SelectCell'
   CellName
'PutCell'
   U4for1

CellName = 'V4'
'SelectCell'
   CellName
'PutCell'
   V4for1

CellName = 'W4'
'SelectCell'
   CellName
'PutCell'
   W4for1

TopCorner = 'U4'
BottomCorner ='W'||LastSplit
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calculate'
```

→

```
'macro'
  macro

CellName = 'O'||SplitRow
'SelectCell'
   CellName
'PutCell'
   "split 4/1"

RETURN


BUILD_3FOR2:
/* Process 3 for 2 split.                                    */

CellName = 'U4'
'SelectCell'
   CellName
'PutCell'
   U3for2

CellName = 'V4'
'SelectCell'
   CellName
'PutCell'
   V3for2

CellName = 'W4'
'SelectCell'
   CellName
'PutCell'
   W3for2

TopCorner = 'U4'
BottomCorner ='W'||LastSplit
'SelectRange'
  TopCorner
  BottomCorner
macro = 'fill_down'
'macro'
  macro

macro = 'calculate'
'macro'
  macro

CellName = 'O'||SplitRow
'SelectCell'
   CellName
'PutCell'
   "split 3/2"

RETURN


FIND_HIGH_LOW:
/* Find the all time high and low close.                     */

CellName = 'D'||CurrentRow
'SelectCell'
   CellName
'GetValue'
Value = result

IF Value > AllTimeHigh
THEN
   DO
   AllTimeHigh = Value
   CellName = 'A'||CurrentRow
   'SelectCell'
      CellName
   'GetValue'
   Value = result
   AllTimeHighDate = Value
   END
ELSE
   IF Value < AllTimeLow
   THEN
      DO
      AllTimeLow = Value
      CellName = 'A'||CurrentRow
      'SelectCell'
         CellName
      'GetValue'
      Value = result
      AllTimeLowDate = Value
            END
   ELSE
      NOP

CurrentRow = CurrentRow + 1

RETURN
```

Listing Ten

```
/*********************************************************/
/*                        Open                           */
/*                                                       */
/* This program graphs the data in a spreadsheet created */
/* by StoxUpdate. Arguments to the program are:          */
/*                                                       */
/*      stock symbol                                     */
/*      number of days to graph                          */
/*                                                       */
/* The number of days is from the most recent day        */
/* backwards. If not entered it defaults to all the data. */
/*                                    Jack Fox           */
/*********************************************************/

ARG Symbol Days

OPTIONS results

CALL PRAGMA('Directory','T:')

PlanZoo = 'STOX:'||Symbol||'.zoo'
PlanName = 'T:'||Symbol||'.adv'

AlreadyLoaded = N

/* See if we have graphed another stock.                  */
wk1 = GETCLIP('Last')
LastAdv = 'T:'||wk1||'.adv'

/* See if the spreadsheet we want to graph is already     */
/* unZOOed in the work area. If another spreadsheet has   */
/* been graphed, we need to close its graph widows first. */
IF EXISTS(PlanName)
   THEN
   IF EXISTS(LastAdv)
      THEN
      DO
            AlreadyLoaded = Y
      CALL CLOSE_WINDOWS
            END
   ELSE
      NOP
ELSE
   IF EXISTS(PlanZoo)
      THEN
            DO
            IF EXISTS(LastAdv)
               THEN
                     DO
                     ADDRESS COMMAND 'delete' LastAdv
         CALL CLOSE_WINDOWS
         END
            ELSE
               NOP
      ADDRESS COMMAND 'zoo e' PlanZoo
            SETCLIP('Last',Symbol)
            END
   ELSE
      DO
      SAY PlanZoo||' does not exist'
      EXIT
      END

/* Edit the "days" argument.                              */
IF Days = ''
   THEN
   NOP
ELSE
   DO
   CALL DATATYPE(Days)
   WkType = Result
   IF WkType = 'NUM'
      THEN
            DaysFlag = 'Y'
   ELSE
      DO
            SAY 'invalid days parameter'
            EXIT
            END
   END

ADDRESS 'Advantage'

/* If we are re-graphing a spreadsheet that has just been */
/* graphed, we don't need to reload it.                   */
IF AlreadyLoaded = Y
THEN
   NOP
ELSE
   DO
   'LoadSpread'
     PlanName
   END

/* Determine the last row in the spreadsheet.             */
```

```
macro = 'select_all'
'macro'
  macro
'current'
  temp = result

PARSE VAR temp wk1 ':' wk2
CALL SUBSTR(wk2,2)
LastRow = result

/* If we aren't graphing all the data, determine where to */
/* start graphing from.                                   */
IF DaysFlag = 'Y'
   THEN
   DO
   WkDays = LastRow - Days
   RowStart = WkDays + 1
   IF RowStart < 4
      THEN
      DO
            SAY 'invalid days parameter'
      RowStart = 4
            END
   ELSE
      IF RowStart > LastRow
               THEN
               DO
                     SAY 'invalid days parameter'
                     RowStart = 4
                     END
      ELSE
               NOP
   END
ELSE
   RowStart = 4

/* Use Scripit to bring the spreadsheet screen up.        */
ADDRESS XIT
'WAIT 25'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SC FRONT'
'WAIT 25'

ADDRESS 'Advantage'

/* High-Low chart of high, low, and close.                */
TopCorner = 'B'||RowStart
BottomCorner ='D'||LastRow
'SelectRange'
  TopCorner
  BottomCorner

macro = 'chart_1'
'macro'
  macro

/* Resize the chart just drawn.                           */
ADDRESS XIT
'WAIT 5'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
'W RESIZETO 640,120'
'WAIT 5'

ADDRESS 'Advantage'

/* Line chart of close, twenty, and fifty day moving      */
/* average of closing price.                              */
TopCorner = 'D'||RowStart
BottomCorner ='F'||LastRow
'SelectRange'
  TopCorner
  BottomCorner

macro = 'chart_2'
'macro'
  macro

/* Resize the chart just drawn.                           */
ADDRESS XIT
'WAIT 25'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
'W RESIZETO 640,120'
'WAIT 5'

ADDRESS 'Advantage'

/* Floating bar chart of percent of day's volume above or */
/* below the fifty day moving average of the volume.      */
TopCorner = 'N'||RowStart
BottomCorner ='N'||LastRow
'SelectRange'
  TopCorner
  BottomCorner

macro = 'chart_3'
'macro'
  macro

/* Resize and move the chart just drawn.                  */
ADDRESS XIT
'WAIT 5'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
'W RESIZETO 640,120'
'WAIT 5'
'W MOVE 0 95'
'WAIT 5'


ADDRESS 'Advantage'

/* Line chart of twenty and fifty day moving average of   */
/* ratio between volume on "up" days and "down" days.     */
TopCorner = 'K'||RowStart
BottomCorner ='L'||LastRow
'SelectRange'
  TopCorner
  BottomCorner

macro = 'chart_4'
'macro'
  macro

/* Resize and move the chart just drawn.                  */
ADDRESS XIT
'WAIT 5'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
'W RESIZETO 640,120'
'WAIT 5'
'W MOVE 0 190'
'WAIT 5'

ADDRESS 'Advantage'

/* Line chart of five day moving average of ratio between */
/* volume on "up" days and "down" days.                   */
TopCorner = 'J'||RowStart
BottomCorner ='J'||LastRow
'SelectRange'
  TopCorner
  BottomCorner

macro = 'chart_5'
'macro'
  macro

/* Resize and move the chart just drawn.                  */
ADDRESS XIT
'WAIT 5'
'SELECT SCREEN "The Advantage, V1.1. By Michal"'
'SELECT WINDOW "PlanName"'
'W ACTIVATE'
'W RESIZETO 640,120'
'WAIT 5'
'W MOVE 0 300'

EXIT


CLOSE_WINDOWS:
/* Close chart windows previously drawn.                  */

ADDRESS XIT
'SELECT SCREEN "The Advantage, V1.1. By Michal" '
'SC FRONT'
'W ACTIVATE'
'W CLOSE'
'WAIT 100'
'SELECT SCREEN "The Advantage, V1.1. By Michal" '
'W ACTIVATE'
'W CLOSE'
'WAIT 100'
'SELECT SCREEN "The Advantage, V1.1. By Michal" '
'W ACTIVATE'
'W CLOSE'
'WAIT 100'
'SELECT SCREEN "The Advantage, V1.1. By Michal" '
'W ACTIVATE'
'W CLOSE'
'WAIT 125'
'SELECT SCREEN "The Advantage, V1.1. By Michal" '
'W ACTIVATE'
'W CLOSE'
'WAIT 100'

RETURN
```

Collectible Disks!

The Fred Fish Collection

Choose from the entire Fred Fish collection and get your disks quickly and easily by using our toll free number: **1-800-345-3360**.

Our collection is updated constantly so that we may offer you the best and most complete selection of Fred Fish disks anywhere.

Now Over 530 Disks!

Disk prices for *AC's TECH* subscribers:

1 to 9 disks - $6.00 each
10 to 49 disks - $5.00 each
50 to 99 disks - $4.00 each
100 disks or more - $3.00 each

(Disks are $7.00 each for non-subscribers)

You are protected by our no-hassle, defective disk return policy*

To get **FAST SERVICE** on Fred Fish disks, use your Visa or MasterCard and

call 1-800-345-3360.

Or, just fill out the order form on page 48.

* *AC's TECH* warranties all disks for 90 days. No additional charge for postage and handling on disk orders. *AC's TECH* issues Mr. Fred Fish a royalty on all disk sales to encourage the leading Amiga program anthologist to continue his outstanding work.

Language Extensions
Strings of Type StringS

by Jimmy Hammonds

Extending a programming language beyond the "standard" is something that is done by nearly all compiler software vendors. This practice is normally used to take advantage of the unique capabilities of the targeted computer hardware. Since the extensions are specific to a particular programming language, rarely will you see one programming language's extension apply to another programming language as well—that is, until now. With the implementation of strings of type StringS, all programming languages that can access the Amiga's disk-based libraries have been extended to include this new data type. StringS's size and structure are the same across all programming languages. The functions used to manipulate them are the same as well. Just define the StringS data type. Create a variable of that type. Open the support library and you're off to the races.

StringS is an enhanced string type. It is a pointer to a structure describing the string's data. This new type is not alone; it comes with a supporting cast of library functions to show it off. These strings are very flexible. They grow and shrink and consume only the amount of memory required to contain them. The structure pointed to by a StringS variable contains three members, ss_String, ss_Length, and ss_Mempool. ss_String is a pointer to the actual characters and can be any character, including the NULL (\0) character. ss_Length is the count of characters defining ss_String. And ss_Mempool is the address of a memory block to which the memory allocated to this string belongs. More on the memory pool later. The library that contains the support functions is a shared library—that is, a non-linking library. This is the reason multiple programming languages can make use of this new data type. StringS are created, manipulated, and deleted by the appropriate support functions. They can also have their contents changed by standard C constructs. This new string type, with its supporting cast of library functions, provides some unique features that do not exist in most programming languages that I know of for the Amiga computer.

Sample Program

Since C is my programming language of choice, with assembler a very close second, we will be discussing the implementation of strings of type StringS using C constructs. The C compiler used for developing and testing this string implementation is Aztec C 5.0a. Arithmetic expression evaluator (Aee), a sample program which makes use of most of the support functions, is provided. This program accepts and resolves arithmetic expressions. Enhance this program if you like. I have found I learn best by doing than by reading alone. To compile this program, you must compile with the 32-bit integer option and use the precompiled header file PreCompiledHeader. If you do not have Aztec's compiler, you can create your own compiled header file by compiling the file PreCompiledHeader.c. You must also use the floating point version of the math link library in the link. If your compiler cannot accept #pragma statements, uncomment the line that reads "#define string_glue" and include the link library file ss.lib in the link.

Type StringS

Type StringS is a type definition and it looks like this:

```
typedef struct StringS {
                          BYTE *const ss_String;

                        }*StringS;
```

Most of the supporting functions accept and return strings of this type. A NULL string is a StringS variable with a value of zero. ss_String is a pointer to the characters comprising the string. This pointer address must not be changed. Note that ss_Length? and ss_Mempool are missing from this structure. These two items are for use by the support library, and are not to be referenced by the calling program.

Support Library

StringSupport.library is the shared library program containing the support functions. To use this library, you must declare a structure pointer of type StringBase and call the Amiga's OpenLibrary routine. This procedure is no different than opening one of the Amiga's disk-based libraries (i.e., Icon, Info). The StringBase structure looks like this:

```
struct StringBase {
                    Library sb_Library;
                    struct ExecBase *sb_SysLib;
                    struct Segment *sb_SegList;
                    struct List *sb_Mempool;
                    UBYTE sb_Flags;
                    UBYTE sb_pad;
                    struct SignalSemaphore sb_Signalsem;
                    const StringS sb_ALPHA;
                    const StringS sb_NUMERIC;
                    const StringS sb_ALPHANUMERIC;
                    const StringS sb_SPACE;
                    const StringS sb_PERIOD;
                    const StringS sb_NULL;
                    const StringS sb_SPECIAL;
                    };
```

The relevant parts of this structure are the StringS constants. sb_ALPHA contains all alpha characters upper and lower case, lower case first. sb_NUMERIC contains the digits '0' thru '9'. sb_ALPHANUMERIC contains all characters of sb_ALPHA and sb_NUMERIC. sb_SPACE contains one space character.

sb_PERIOD contains one period character. sb_NULL contains one NULL character. sb_SPECIAL contains the characters "!@#$%^"&*()_+|-=\{}[]:;'<>?,./". These StringS constants can be used like any other StringS variable but must not be altered or have their address changed. These constants are global to all programs using the support library.

All functions of this library accepts parameters in registers. With Aztec's C compiler, #pragma statements can be used to define register usage and have the compiler fill them at the time you invoke the function. I understand Lattice C has an equivalent setup, although I have not tried it. The appropriate #pragma statements are as follows:

```
#pragma amicall (StringBase,30,AllocS(d0))
#pragma amicall (StringBase,36,CompareS(a0,a1,d0,d1))
#pragma amicall (StringBase,42,DeallocS(a0))
#pragma amicall (StringBase,48,DropS(a0,d0,d1))
#pragma amicall (StringBase,54,HeadS(a0,a1,d0,d1,d2))
#pragma amicall (StringBase,60,LengthS(a0))
#pragma amicall (StringBase,66,RepeatS(a0,d0,d1))
#pragma amicall (StringBase,72,SearchS(a0,a1,d0,d1))
#pragma amicall (StringBase,78,TailS(a0,a1,d0,d1,d2))
#pragma amicall (StringBase,84,TakeS(a0,d0,d1))
#pragma amicall (StringBase,90,ConcatS(a0,a1,d0,d1))
#pragma amicall (StringBase,96,CopyS(a0,d0))
#pragma amicall (StringBase,102,MakeS(a0))
#pragma amicall (StringBase,108,ConfigS(d0))
#pragma amicall (StringBase,120,AvailS())
#pragma amicall (StringBase,126,
                 TranslateS(a0,a1,a2,d0,d1,d2))
```

Note: The #pragma statements, function, and StringS declarations are in an include file named library/stringsupport.h. I have also included a C .lib file with the appropriate "glue" routines for compilers that cannot pass parameters through registers. The name of this .lib file is ss.lib (sslib.asm).

Library Internals

When a program opens the library, a pool of memory, 5KB, is allocated for use as StringS memory. A memory pool is allocated for each program (task) opening the library. After opening the library and before allocating any memory for a StringS variable, the program can request the allocation of a larger memory pool. If the larger memory pool can not be allocated or memory has been allocated to a StringS variable, the request will be ignored. Each program using the support library functions must open the library first. This is an absolute must, since the address of the program's TASK structure is the link to its memory pool. The Amiga library routines, Allocate and Deallocate, are used to manage this memory pool. Each time a StringS variable is assigned a value via one of the StringS functions, new memory is allocated for its use. If a StringS variable is being reassigned a new value, its old memory space can be returned to the memory pool for reuse. Upon closing the library, the memory pool allocated for StringS memory will be returned to the systems free memory list.

Note: If there is not enough memory available when requesting a new StringS, a NULL StringS is returned.

Installation

Assemble the file Library/stringsupport.asm. Link it. Then copy the object file to the libs: directory on your boot disk as StringSupport.library. StringSupport is now installed.

Supporting Functions

The following variables will be used in the example that follow each function description.

```
StringS work, hello, s1, s2;
LONG cmp;
ULONG size;
BYTE letters[] = "abcdefabcdefg";
BYTE Hello[] = "Hello World, how are ya!";
```

StringS AllocS (ULONG size)

AllocS returns a new string of <size> length. The memory space allocated is not initialized. This is where life begins for a StringS string.

```
ex.             work = AllocS(25);
```

Work is a StringS containing 25 undefined characters. It should be noted here that the maximum StringS length supported is 65535. A request for a length greater than the maximum returns a NULL StringS.

StringS MakeS (BYTE *string)

MakeS returns a new StringS containing the characters pointed to by <string>. <string> must be a pointer to a NULL-terminated character string.

```
ex.             work = MakeS("What is your name?");
```

In this example a StringS is made from a string of NULL-terminated characters.

```
LONG CompareS (StringS stringA, StringS string, LONG deallocA, LONG
deallocB)
```

CompareS compares to StringS and returns a value of 0 if <stringA> is less than <stringB>, a value of 0 if <stringA> is equal to <stringB>, and a value of 1 if <stringA> is greater than <stringB>. A -1 is also returned if <stringA> or <stringB> is a NULL StringS. If <deallocA> is non zero, <stringA> will be deallocated before CompareS returns. If <deallocB> is non zero, <stringB> will be deallocated before CompareS returns.

```
ex.             s1 = TakeS(MakeS(&letters),6,TRUE);
                s2 = DropS(MakeS(&letters),6,TRUE);
                cmp = CompareS (s1,s2,FALSE,FALSE);
                if (cmp < 0) printf ("s1 < s2");
                else if (cmp > 0) printf ("s1 > s2");
                          else printf ("s1 = s2");
```

In the above s1 contains the letters "abcdef" and s2 contains the letters "abcdefg." After executing the CompareS statement, cmp will contain the value -1. s1 is less than s2. The StringS s1 and s2 are not be deallocated through the execution of this statement.

StringS DeallocS (StringS stringA)

DeallocS frees memory allocated to the <stringA>. This StringS variable should not be used again until it has been assigned a new value by one of the StringS functions. This function always returns a NULL StringS. It is very important to deallocate a StringS when it is no longer needed. This function not only frees the memory, it also combines contiguous memory blocks to help prevent memory checkerboarding. This function will also be called by most of the other functions, to deallocate memory, if requested to do so.

```
ex.             work = MakeS(&letters);
                work = DeallocS(work);
```

In this example, work allocated a StringS and then deallocated.

```
            movem.l     (sp)+,a6/d2
            rts

_LengthS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            jsr         LengthS(a6)
            movea.l     (sp)+,a6
            rts

_RepeatS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            move.l      12(sp),d0
            move.l      16(sp),d1
            jsr         RepeatS(a6)
            movea.l     (sp)+,a6
            rts

_SearchS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            movea.l     12(sp),a1
            move.l      16(sp),d0
            move.l      20(sp),d1
            jsr         SearchS(a6)
            movea.l     (sp)+,a6
            rts

_TailS:
            movem.l     a6/d2,-(sp)
            movea.l     _StringBase,a6
            movea.l     12(sp),a0
            movea.l     16(sp),a1
            move.l      20(sp),d0
            move.l      24(sp),d1
            move.l      28(sp),d2
            jsr         TailS(a6)
            movem.l     (sp)+,a6/d2
            rts

_TakeS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            move.l      12(sp),d0
            move.l      16(sp),d1
            jsr         TakeS(a6)
            movea.l     (sp)+,a6
            rts

_ConcatS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            movea.l     12(sp),a1
            move.l      16(sp),d0
            move.l      20(sp),d1
            jsr         ConcatS(a6)
            movea.l     (sp)+,a6
            rts

_CopyS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            move.l      12(sp),d0
            jsr         CopyS(a6)
            movea.l     (sp)+,a6
            rts

_MakeS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            jsr         MakeS(a6)
            movea.l     (sp)+,a6
            rts

_ConfigS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            move.l      8(sp),d0
            jsr         ConfigS(a6)
            movea.l     (sp)+,a6
            rts

_ChunkS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            movea.l     8(sp),a0
            jsr         ChunkS(a6)
            movea.l     (sp)+,a6
            rts

_AvailS:
            move.l      a6,-(sp)
            movea.l     _StringBase,a6
            jsr         AvailS(a6)
            movea.l     (sp)+,a6
            rts

_TranslateS:
            movem.l     a2/a6/d2,-(sp)
            movea.l     _StringBase,a6
            movea.l     16(sp),a0
            movea.l     20(sp),a1
            movea.l     24(sp),a2
            move.l      28(sp),d0
            move.l      32(sp),d1
            move.l      36(sp),d2
            jsr         TranslateS(a6)
            movem.l     (sp)+,a2/a6/d2
            rts
            end
```

precompiledheader.c

```
#include <stdio.h>
#include <ctype.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <math.h>
#include <functions.h>
#include <pragmas.h>

#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <libraries/filehandler.h>

#include <exec/types.h>
#include <exec/io.h>
#include <exec/memory.h>
#include <exec/nodes.h>
#include <exec/ports.h>
#include <exec/tasks.h>
#include <exec/alerts.h>
#include <exec/libraries.h>
#include <exec/errors.h>
#include <exec/semaphores.h>
#include <exec/lists.h>

#include <graphics/gfxbase.h>

#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
```

Please refer to the enclosed disk for the following listings: aee.c, stringsupport.asm, and stringsupport.h. Also included are the above listings.

State of Amiga Development Denver DevCon Address

(continued from page 3)

CATS

CATS is here to help you. That's the only reason we exist. Please take advantage of the services we offer. This year, CATS has released more developer tools and documentation than in any other year. Here are a few examples:

The User Interface Style Guide will help you create more intuitive, easy to use, and standard user interfaces, for your applications.

The Application Installer, available from CATS for free distribution with your applications, provides a standard and simple way for your users to install their applications on their hard disks. Use it.

"In the next month or two, we expect to ship the three-millionth Amiga!"

Jeff Scherb
CATS

AppShell, a development tool designed to serve as a sample application and a basis for the event processing code in your applications, is available now. Take advantage of this tool to speed up your application development.

The ARexx development guide will help you add this important interprocess language capability to your applications.

We now have AmigaGuide, a hypertext development and runtime system, available to you for inclusion in your applications. Using this tool, you can easily add hotkey hyertext-linked help text to your applications. Be sure to attend the session on AmigaGuide to learn all about this new tool.

Most of the 2.0 ROM Kernel Reference Manuals are finished, and either are available now or are in the final printing stages at Addison-Wesley.

How many of you are using Enforcer and Mungwall? Every developer should be using these debugging tools as part of their normal routine. MS-DOS and Mac developers can only dream of powerful tools like this, since they can't be implemented on those primitive computers. Make sure you take advantage of these tools.

Other CATS Projects

As I said before, if you're not in Europe, you need to be. *"Crossing Borders,"* a new reference book to be available from CATS early this fall, takes a "cookbook" approach to getting to Europe. How do you get distribution? Support? Translation of your manuals and programs? *"Crossing Borders"* is the manual that will help you answer these questions. Make sure you get your copy as soon as it's ready.

We're putting the Amiga developer docs on CD-ROM, using our own AmigaGuide hypertext system. Using this, you'll be able to integrate this on-line reference to the Amiga operating system and hardware into your favorite text editor. You can call up OS function definitions while you're programming, and using the cut-and-paste features, move examples directly from the documentation into your code. This should be available sometime this fall, and you can see a demo of it this week in the hardware lab.

We're arranging special Developer/Distributor meetings to help you meet the European distributors and gain distribution in Europe. The first of these will be held this year at Amiga '91 Cologne, the largest Amiga show in the world. Last year, this show drew 60,000 people. You'll be getting a mailing from CATS with information on these meetings next week.

Devcon Highlights

- Several sessions on new hardware;
- detailed CDTV sessions;
- a presentation by a translation service to help you move your applications to Europe;
- several sessions by Commodore's US marketing department, to help you understand where we're going from a marketing point of view in the US;
- a session on finding the right European distributor, by CATS Applications Manager, Europe, Wolfgang Trompetter;
- many other technical hardware and software sessions.

Thank you,

Jeff Scherb
Vice President, CATS

DevCon Overview

The Denver DevCon displayed Commodore's commitment to working closely with the Amiga Developer's Community. Some special areas of interest were seminars on marketing and packaging products, as well as finding distributors and addressing localization issues.

There was certainly a high concentration on development for CDTV. Sessions included development guidelines, packaging, licensing, and a general overview of Commodore's direction. There is a new spirit of commitment from Commodore, for both CDTV and other development areas.

Though specific information regarding new products can not be discussed, by the looks of things, the Amiga community is poised for rapid, successful growth in the coming year. The vast expanse of new items and improvements to existing products will help to make the future of the Amiga profitable for users, developers, and Commodore.

Jeff and Commodore continued their commitment to Amiga developers the following week in Milan, Italy with a European DevCon. The energy displayed by Commodore combined with the enthusiasm from developers will make this a great time to be involved with the Amiga.— Jeff Gamble, Amazing Computing

StringS DropS (StringS stringA, LONG dropcount, LONG deallocA)

DropS returns a new StringS of remaining characters after dropping <dropcount> number of characters from <stringA>. If all characters are dropped, a NULL StringS is returned. If <deallocA> is non zero, <stringA> will be deallocate before DropS returns. If <dropcount> is zero, DropS returns a copy of <stringA>.

```
ex.                hello = MakeS(&Hello);
                   work = DropS (DropS (hello,13,FALSE),8,TRUE);
```

work will be a StringS containing "ya!" when the functions complete. Note the last parameter of TRUE for the outer DropS function. This is required to return the memory, allocated for the results of the inner DropS function, to the memory pool since it is no longer needed.

```
StringS HeadS (StringS stringA, StringS stringSET
               ,LONG deallocA, LONG deallocSET, LONG notSET)
```

HeadS returns a new StringS containing all leading characters from <stringA> that are present in <stringSET>. If <notSET> is non zero, HeadS returns a new StringS? containing all leading characters from <stringA> that are not present in <stringSET>. If no match is found, a NULL StringS is returned. If <deallocA> is non zero, <stringA> will be deallocated before HeadS returns. If <deallocSET> is non zero, <stringSET> will be deallocated before HeadS returns.

```
ex.                hello = MakeS(&Hello);
                   ?hello = HeadS(hello,StringBase->sb_ALPHA,TRUE,FALSE);
```

In this example, the StringS hello contains the characters "Hello" after completion of the function. Only leading alpha characters are being taken from the StringS. Note that the memory previously allocated to the StringS hello is being returned to the memory pool.

```
ULONG LengthS (StringS stringA)
```

LengthS returns the length of <stringA>.

```
ex.                s1 = MakeS(&letters);
                   printf ("s1's length is %ld\n",LengthS(s1))
```

This example prints out the length of the StringS s1.

```
StringS RepeatS (StringS stringA, LONG repeatcount
                 ,LONG deallocA)
```

RepeatS returns a new StringS containing <stringA> repeated <repeatcount> times. If <repeatcount> is zero, a NULL StringS is returned. If <deallocA> is non zero, <stringA> will be deallocated before RepeatS returns.

```
ex.                s1 = MakeS("*");
                   s2 = RepeatS(s1,80);
```

This example will make a StringS of one asterisk. Then create a new StringS containing 80 asterisks.

```
LONG SearchS (StringS stringA, StringS stringB
              ,LONG deallocA, LONG deallocB)
```

SearchS searches the StringS <stringB> for the first occurrence of <stringA>. It then returns the character position from the beginning of <stringB> where <stringA> was found. If <stringA> is not found, a -1 is returned. A -1 is also returned if <stringA> or <stringB> is a NULL StringS. If <deallocA> is non zero, <stringA> will be deallocated before SearchS returns. If <deallocB> is non zero, <stringB> will be deallocated before SearchS returns.

```
ex.                hello = MakeS(&Hello);
                   printf("position of the word 'How' is %ld\n"
                            ,SearchS(MakeS("How"),hello,TRUE,FALSE));
```

This example prints out the relative position of the word "How" in the StringS hello. If the word cannot be found, a negative one is printed for the position. In this case 13 is printed.

```
StringS TailS (StringS stringA, StringS stringSET
               ,LONG deallocA, LONG deallocSET, LONG notSET)
```

TailS returns a new StringS of all remaining characters after removing all leading characters from <stringA> that are present in <stringSET>. If <notSET> is non zero, TailS returns a new StringS of all remaining characters after removing all leading characters from <stringA> that are not present in <stringSET>. If no match is found, a NULL StringS is returned. If <deallocA> is non zero, <stringA> will be deallocated before TailS returns. If <deallocSET> is non zero, <stringSET> will be deallocated before TailS returns.

```
ex.                hello = MakeS(&Hello);
                   hello = TailS(hello,StringBase->sb_ALPHA,TRUE,FALSE);
```

In this example, the StringS hello contains the characters "World, How are ya!" after execution of this function. Only leading alpha characters have been dropped from the StringS. Note that the memory previously allocated to the StringS hello is being returned to the memory pool.

```
StringS TakeS (StringS stringA, LONG takecount
                ,LONG deallocA)
```

TakeS returns a new StringS containing <takecount> leading characters from <stringA>. If <takecount> is zero, a NULL StringS is returned. If <deallocA> is non zero, <stringA> will be deallocated before TakeS returns.

```
ex.                work = TakeS(StringBase->sb_ALPHA,26,FALSE);
```

In this example, work contains all the lowercase letters from the StringS constant sb_ALPHA when the function completes. Since this StringS is a constant provided by the library, it must not be deallocated.

```
StringS ConcatS (StringS stringA, StringS stringB
                 ,LONG deallocA, LONG deallocB)
```

ConcatS returns a new StringS containing the combination of <stringA> and <stringB>. If <deallocA> is non zero, <stringA> will be deallocated before ConcatS returns. If <deallocB> is non zero, <stringB> will be deallocated before ConcatS returns.

```
ex.                work = ConcatS(StringBase->sb_NUMERIC
                                  ,StringBase->sb_PERIOD
                                  ,FALSE,FALSE);
```

This example combines two StringS into one. Upon completion of the function, work will contain the numbers 0 thru 9 and the ".". The two constant StringS are not deallocated.

StringS CopyS (StringS stringA, LONG deallocA)

CopyS returns a new StringS containing all characters from <stringA>. If <deallocA> is non zero, <stringA> will be deallocated before CopyS returns.

```
ex.                 work = CopyS(StringBase->sb_SPECIAL,FALSE);
```

This example returns to the StringS work a copy of the sb_SPECIAL constant.

ULONG ConfigS (ULONG size)

ConfigS allows a program to request more than the default 5KB for the StringS memory. <size> is the amount of memory to allocate for the memory pool. This request must be made before any StringSs are allocated. If the request is made after StringSs have been allocated or there is insufficient memory available, the request will be ignored. In either case the amount of memory in the pool will be returned.

```
ex.                 size = Config (1024*512);
```

After the execution of this function, the new StringS memory pool size will be 0.5MB unless there was insufficient memory or memory already allocated.

ULONG AvailS()

AvailS returns the amount of free StringS memory.

```
ex.                 size = AvailS();

StringS TranslateS (StringS stringA, StringS stringB
                          ,StringS stringC, LONG deallocA
                          ,LONG deallocB, LONG deallocC)
```

TranslateS takes characters from <stringA>, compares them to characters in <stringB>, and replaces them with characters from <stringC> when a match is found. TranslateS operates in one of two modes. In the first mode <stringB> and <stringC> must be the same size and multiple characters can be replaced. In the second mode <stringC> must have a length of one, and all characters found in <stringB> are replaced with the one character from <stringC>. If <deallocA> is non-zero, <stringA> will be deallocated before TranslateS returns. If <deallocB> is non-zero, <stringB> will be deallocated before TranslateS returns. If <deallocC> is non zero, <stringC> will be deallocated before TranslateS returns.

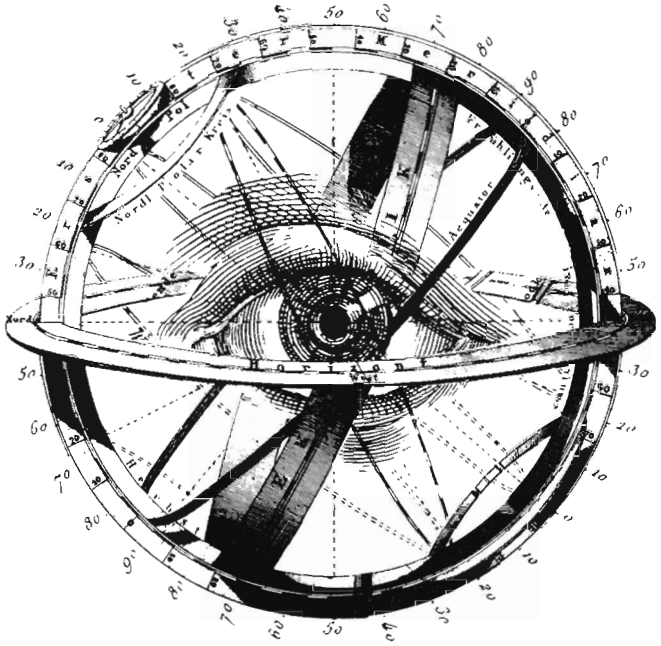
```
ex.                 work = TranslateS(MakeS("abcdefg");
                          ,TakeS(StringBase->sb_ALPHA,26,FALSE)
                          ,DropS(StringBase->sb_ALPHA,26,FALSE)
                          ,TRUE
                          ,TRUE
                          ,TRUE
                          );
                    printf ("work = %.*s\n",Length(work),work->ss_String);
```

In this example, the letters "abcdefg" are translated from lower-case to uppercase.

sslib.asm

```
;
;          Copyright (c) 1990 by Jimmy L. Hammonds
;          All Rights Reserved
;
 include "exec/types.i"
 include "exec/nodes.i"
 include "exec/lists.i"
 include "exec/libraries.i"
 include "exec/alerts.i"
 include "exec/resident.i"
 include "exec/initializers.i"
 LIBINIT
 LIBDEF    AllocS
 LIBDEF    CompareS
 LIBDEF    DeallocS
 LIBDEF    DropS
 LIBDEF    HeadS
 LIBDEF LengthS
 LIBDEF RepeatS
 LIBDEF SearchS
 LIBDEF TailS
 LIBDEF TakeS
 LIBDEF ConcatS
 LIBDEF CopyS
 LIBDEF MakeS
 LIBDEF ConfigS
 LIBDEF ChunkS
 LIBDEF AvailS
 LIBDEF TranslateS
;
 XDEF      _AllocS
 XDEF      _CompareS
 XDEF      _DeallocS
 XDEF      _DropS
 XDEF      _HeadS
 XDEF      _LengthS
 XDEF      _RepeatS
 XDEF      _SearchS
 XDEF      _TailS
 XDEF      _TakeS
 XDEF      _ConcatS
 XDEF      _CopyS
 XDEF      _MakeS
 XDEF      _ConfigS
;XDEF      _ChunkS
 XDEF      _AvailS
 XDEF      _TranslateS
;
 XREF      _StringBase
;
_AllocS:
          move.l    a6,-(sp)
          movea.l   _StringBase,a6
          move.l    8(sp),d0
          jsr       AllocS(a6)
          movea.l   (sp)+,a6
          rts

_CompareS:
          move.l    a6,-(sp)
          movea.l   _StringBase,a6
          movea.l   8(sp),a0
          movea.l   12(sp),a1
          move.l    16(sp),d0
          move.l    20(sp),d1
          jsr       CompareS(a6)
          movea.l   (sp)+,a6
          rts

_DeallocS:
          move.l    a6,-(sp)
          movea.l   _StringBase,a6
          movea.l   8(sp),a0
          jsr       DeallocS(a6)
          movea.l   (sp)+,a6
          rts

_DropS:
          move.l    a6,-(sp)
          movea.l   _StringBase,a6
          movea.l   8(sp),a0
          move.l    12(sp),d0
          move.l    16(sp),d1
          jsr       DropS(a6)
          movea.l   (sp)+,a6
          rts

_HeadS:
          movem.l   a6/d2,-(sp)
          movea.l   _StringBase,a6
          movea.l   12(sp),a0
          movea.l   16(sp),a1
          move.l    20(sp),d0
          move.l    24(sp),d1
          move.l    28(sp),d2
          jsr       HeadS(a6)
```

SAY REVOLUTION



The fastest growing video technology company in the world is looking for programmers to join our team. We've assembled the hottest development group in the industry here at NewTek. But we have two slots open for a new project that will blow your socks off. Have you always dreamed of working on revolutionary technology in a small, focused group? We need software innovators that want to create the products of tomorrow. Here are the skills you'll need:

- Strong 68xxx assembly-language programming skills
- At least 3 years of assembly-language programming experience
- Ability to write low-level code for time-critical applications
- Background in high-speed graphics and video applications
- Experience in programming prototype hardware
- Ability to quickly learn new custom chip architectures
- Background in low-level I/O and interrupt operations
- Intimate understanding of Amiga O/S and hardware
- Experience with video and graphics hardware
- Ability to read a schematic
- Strong organizational and project design skills
- Being a self-motivated, self-teaching, innovator
- An uncompromising drive for excellence

If you've got what it takes, you'll be forging ahead where no programmer's ever gone before. NewTek offers outstanding (and unusual), compensation and benefit packages for the chosen few who are a cut above. You'll work in an environment created by hackers, designed to be a hackers heaven. Wouldn't it be fun to invent things that are featured in USA Today, Rolling Stone, and TIME? At NewTek your brain can change the world.

Send Resumes to:

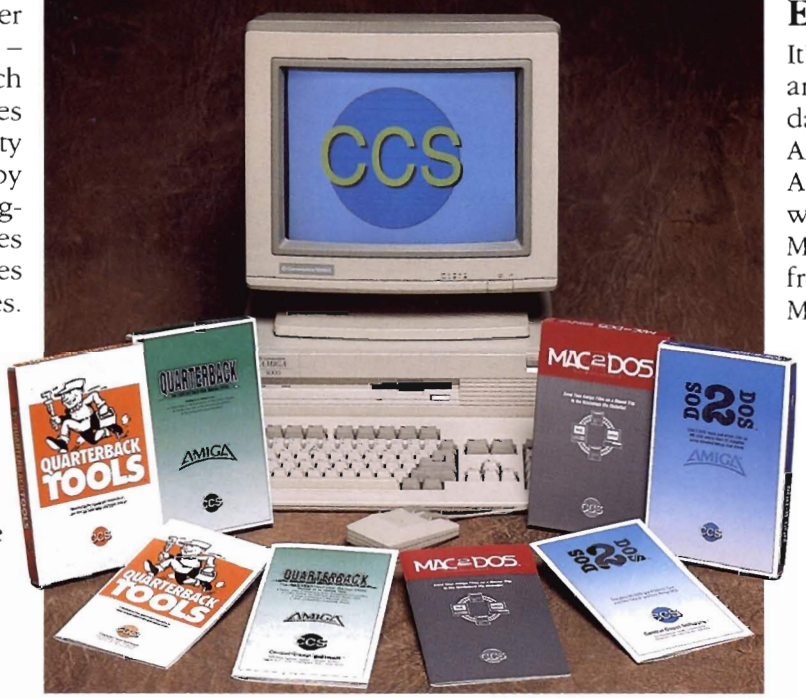
Alcatraz
C/O NewTek
215 SE 8th St.
Topeka, KS 66603

NEWTEK
INCORPORATED

Circle 115 on Reader Service card.

SAVE IT. MOVE IT. GET IT BACK.

Back-Up...Transfer...Retrieve Quickly And Easily With Central Coast's Software For The Amiga



Valuable utility programs can save you time, money and, in the case of catastrophic errors like hard drive failure, possibly months of work.

Quarterback Tools – Recover Lost Files

Fast and easy. Reformats all types of disks – either new or old filing systems – new or old Workbench versions. Also optimizes the speed and reliability of both hard and floppy disks. Eliminates file fragmentation. Consolidates disk space. Finds and fixes corrupted directories.

Quarterback – The Fastest Way To Back-Up

Backing-up has never been easier. Or faster. Back-up to, or restore from: floppy disks, streaming tape (AmigaDOS-compatible), Inner-Connection's Bernoulli drive, or ANY AmigaDOS-compatible device.

Mac-2-Dos & Dos-2-Dos – A Moving Experience

It's easy. Transfer MS-DOS and ATARI ST text and data files to-and-from AmigaDOS using the Amiga's own disk drive with Dos-2-Dos; and Macintosh files to-and-from your Amiga with Mac-2-Dos. Conversion options for Mac-2-Dos include ACSII, No Conversion, MacBinary, PostScript, and MacPaint to-and-from IFF file format.

CCS

Central Coast Software
A Division Of New Horizons Software, Inc.
206 Wild Basin Road, Suite 109, Austin, Texas 78746
(512) 328-6650 * Fax (512) 328-1925

Quarterback Tools, Quarterback, Dos-2-Dos and Mac-2-Dos are all trademarks of New Horizons Software, Inc.

Circle 101 on Reader Service Card